MASTER EN SCIENCES INFORMATIQUES

Définition et implémentation d'un langage graphique pour la description d'assertions

Detiège, Olivier; Zébier , Xavier

Award date:
1998

FACULTÉS UNIVERSITAIRES NOTRE-DAME DE LA PAIX
INSTITUT D'INFORMATIQUE
RUE GRANDGAGNAGE, 21
B-5000 NAMUR

Définition et implémentation d'un langage graphique pour la description d'assertions

Olivier DETIÈGE, Xavier ZÉBIER

Mémoire présenté pour l'obtention du grade
de licencié en informatique

Année académique 1997 - 1998

Merci à notre promoteur Monsieur Baudouin Le Charlier pour les conseils judicieux qu'il nous a prodigué durant l'année.

Je tiens à remercier tous ceux qui ont contribué à la réalisation de ce mémoire ainsi que toute personne m'ayant apporté son soutien.

Olivier Detiège

Je remercie mes parents, mes amis, ainsi que tous ceux qui ont participé de quelque manière que ce soit à la rédaction de ce mémoire.

Xavier Zébier

Table des matières

Introduction 7

1 La méthode proposée 8
- 1.1 Méthode enseignée . . . 8
- 1.2 Démarche . . . 9
  - 1.2.1 Etape 1 : Situation initiale et finale . . . 9
  - 1.2.2 Etape 2 : Situation générale, invariant, condition d'arrêt 10
  - 1.2.3 Etape 3 : Choix des instructions . . . 10
  - 1.2.4 Etape 4 : Vérification de la terminaison . . . 12
  - 1.2.5 Etape 5 : Ecriture . . . 12
- 1.3 Langage de description des schémas . . . 12
  - 1.3.1 Le concept de schéma . . . 12
  - 1.3.2 Le concept de segment . . . 12
  - 1.3.3 Exemple de description d'une situation . . . 14
- 1.4 Description de la méthode . . . 15
  - 1.4.1 Spécifications . . . 15
  - 1.4.2 Situation générale et condition d'arrêt . . . 16
  - 1.4.3 Choix des instructions . . . 16

2 Le cahier des charges **17**
- 2.1 Spécification générale . . . 17
- 2.2 Spécification de la fonctionnalité de manipulation d'objets . . 18
- 2.3 Spécification de la fonctionnalité graphique . . . 18
- 2.4 Spécification de la fonctionnalité de manipulation de code . . . 20
- 2.5 Spécification de la fonctionnalité de condition d'arrêt . . . 20
- 2.6 Fonctionnalité d'enchaînement . . . 20

3 Un langage de saisie des situations **22**
3.1 Syntaxe abstraite 23
3.2 L'environnement et le store 26
3.3 La sémantique 27
3.3.1 Sémantique d'une expression entière 27
3.3.2 Sémantique d'une fonction entière 29
3.3.3 Sémantique d'une expression booléenne 30
3.3.4 Sémantique d'une constante booléenne 32
3.3.5 Sémantique d'un opérateur relationnel 32
3.3.6 Sémantique d'une fonction booléenne 32
3.3.7 Sémantique d'une suite de conditions 34
3.3.8 Sémantique d'une condition 35
3.3.9 Sémantique d'un segment 36
3.3.10 Sémantique d'un bloc 37
3.3.11 Sémantique d'une représentation 39
3.3.12 Sémantique d'une suite de représentations 39
3.3.13 Sémantique d'un tableau 40
3.3.14 Sémantique d'une suite de tableaux 40
3.3.15 Sémantique d'une situation 40
3.3.16 Sémantique d'une description 41
3.3.17 Limitations 41
3.4 Syntaxe concrète 41
3.4.1 Description de la syntaxe 42
3.4.2 Exemples d'utilisation 43

4 Guide d'utilisation **48**
4.1 Le menu principal 48
4.2 Les fonctionnalités 50
4.2.1 Fonctionnalité "Ajout d'un tableau" 51
4.2.2 Fonctionnalité "Ajout d'une représentation" 52
4.2.3 Fonctionnalité "Ajout d'une borne" 53
4.2.4 Fonctionnalité "Ajout d'une variable" 54
4.2.5 Fonctionnalité "Ajout d'une condition" 55
4.2.6 Fonctionnalité "Suppression d'un tableau" 56
4.2.7 Fonctionnalité "Suppression d'une représentation" . . . 57
4.2.8 Fonctionnalité "Suppression d'une borne" 58
4.2.9 Fonctionnalité "Suppression d'une variable" 58
4.2.10 Fonctionnalité "Suppression d'une condition" 59

4.2.11 Fonctionnalité "Initialisation des variables" 60
4.2.12 Fonctionnalité "Initialisation des tableaux" 61
4.2.13 Fonctionnalité "Visualisation des variables" 61
4.2.14 Fonctionnalité "Visualisation des tableaux" 62
4.2.15 Fonctionnalité "Visualisation des conditions" 64
4.3 Lien avec la méthode des invariants 64

5 Exemples d'utilisation **66**
5.1 Exemple 1 : Calcul du carré d'un entier 66
5.1.1 Enoncé du problème 66
5.1.2 Spécifications 66
5.1.3 Situation générale et condition d'arrêt 68
5.1.4 Les différentes suites d'instructions 68
5.1.5 Résultats 71
5.2 Exemple 2 : Calcul de la somme des éléments d'un tableau . . 71
5.2.1 Enoncé du problème 71
5.2.2 Spécifications 71
5.2.3 Situation générale et condition d'arrêt 73
5.2.4 Les différentes suites d'instructions 74
5.2.5 Résultats 74
5.3 Exemple 3 : Calcul du plus long plateau 75
5.3.1 Enoncé du problème 75
5.3.2 Spécifications 75
5.3.3 Situation Générale et condition d'arrêt 77
5.3.4 Les différentes suites d'instructions 77
5.3.5 Résultats 78
5.3.6 Exemple fautif 78

6 Implémentation **82**
6.1 Arbre syntaxique des situations 82
6.1.1 Structure de données 82
6.1.2 Principes généraux 86
6.2 Arbre syntaxique des expressions et instructions 86
6.3 Sémantique des situations et instructions 89
6.4 Evaluation de Delphi 91

7 Les critiques **92**
7.1 Fonctions de sauvegarde 92
7.2 Interface pour la saisie des expressions 92
7.3 Validation du code 92
7.4 Interface graphique 93
7.5 Initialisation automatique de contenu 93
7.6 Bibliothèque de problèmes 93

Conclusion **94**

Introduction

Résumé

Pour obtenir des programmes de bonne qualité, il est généralement admis qu'il est nécessaire d'utiliser des méthodes de construction. Ainsi, aux Facultés Universitaires de Namur, la méthode de l'invariant est enseignée. La construction d'un programme est basée sur la description des situations de celui-ci. Une situation est l'expression de conditions sur l'état des variables et tableaux du programme à un moment déterminé de son exécution. Dans ce mémoire, nous présentons d'abord la méthode et ensuite, le programme d'aide à l'apprentissage de la programmation basé sur celle-ci. Nous l'avons implémenté sous Delphi 2.0. Il vérifie, entre autres, la validité et l'enchaînement des situations. Ce mémoire comporte également la définition et l'implémentation d'un langage graphique permettant la saisie de ces situations.

Abstract

To obtain high quality programs, everyone agrees that construction methods are necessary. Such a method is teached at the Facultés Universitaires de Namur : "the invariant method". The process of program construction is based on the description of its situations. A situation expresses conditions about the variables and arrays at a given stage of the program. In this thesis, the method is first described and then the system based on it is presented. It was written with Delphi 2.0. It checks the correctness and the sequence of the situations. This thesis also includes the definition and the implementation of a graphical language allowing the encoding of these situations.

Chapitre 1

La méthode proposée

1.1 Méthode enseignée

Contrairement à ce qui est couramment admis, la connaissance d'un langage de programmation n'est pas suffisante pour savoir programmer de manière efficace et correcte. A cette connaissance doit s'ajouter celle d'une méthode rigoureuse de construction de programmes, afin de structurer le raisonnement et l'enchaînement des instructions.

La méthode que nous allons développer se base sur l'article [1] de Messieurs Le Charlier et Derroitte. Il utilise :

1. le concept d'objet,
2. le concept de situations initiale et finale,
3. le concept de situation générale et d'invariant,
4. un langage graphique de description de situations.

Plus pragmatiquement, le but étant d'enseigner une méthode de programmation, l'apprentissage de cette dernière se basera sur des problèmes. Comme il ne s'agit pas de tester l'aptitude de l'étudiant à comprendre un énoncé, les problèmes abordés lors de l'enseignement seront donc relativement simples. De même, les structures de données qu'ils aborderont se limiteront donc à

- des constantes entières ou booléennes,
- des variables simples entières ou booléennes,
- des tableaux d'entiers à une dimension.

Le principal avantage du raisonnement par situation est qu'il permet une découpe du programme en un nombre d'entités limités. Il est en effet utopique

de croire l'esprit humain capable de manipuler la globalité d'un problème d'une complexité raisonnable ou élevée. Malheureusement, les étudiants ont tendance à vouloir produire du code directement. Pour ce faire, il essaie de simuler le fonctionnement du programme. Il s'agit de la méthode "essayer-jeter". Ils n'arrivent pas à raisonner en terme de situation. Cette méthode devrait leur permettre d'encoder tout en les obligeant à se baser sur un raisonnement.

1.2 Démarche

Cette démarche peut être décomposée en cinq étapes principales :

1. Situation initiale et finale
2. Situation générale, invariant, condition d'arrêt
3. Choix des instructions
4. Vérification de la terminaison
5. Ecriture

Nous expliquons chacune de ces étapes dans les paragraphes qui suivent.

1.2.1 Etape 1 : Situation initiale et finale

A la réception de l'énoncé du problème, l'étudiant dégage les spécifications du programme. Il exprime le plus clairement possible ce que le programme attend comme données et les résultats qu'il doit produire. Il le fait via les objets utilisés, la situation initiale et la situation finale.

Plus clairement, les **objets utilisés** sont décomposés en deux sous-ensembles :

- Les objets principaux qui sont les données et les résultats du programme ;

  Exemple : un tableau d'entiers à trier

- Les objets auxiliaires qui, eux, sont essentiellement des variables de travail.

  Exemple : une variable de compteur d'itérations

La **situation initiale** (notée *S.I.*) décrit l'ensemble des conditions que devront vérifier les objets utilisés pour garantir que l'exécution du programme ait un sens.

Exemple : l'entier a doit être positif

La **situation finale** (notée *S.F.*) décrit l'ensemble des conditions vérifiées par les objets utilisés après l'exécution du programme.

Exemple : l'entier a est le carré de b

1.2.2 Etape 2 : Situation générale, invariant, condition d'arrêt

La **situation générale** n'a de sens que dans le cas de programmes itératifs. Il s'agit de décrire les propriétés qui sont vérifiées par les objets lors de chaque itération.

Exemple : un indice utilisé pour parcourir un tableau doit toujours se trouver entre la borne inférieure et la borne supérieure de ce tableau

L'**invariant** est la situation générale exprimée sous forme mathématique.

Exemple : $\forall i \ 1 \leq i \leq n \ : \ a[i] > 0$

La **condition d'arrêt** décrit quant à elle, le moment où le programme doit arrêter d'itérer.

Exemple : l'indice parcourant le tableau atteint la borne supérieure de celui-ci

1.2.3 Etape 3 : Choix des instructions

Il s'agit maintenant de décrire les différentes suites d'instructions qui constitueront le corps du programme. Ces suites d'instructions s'articulent autour de la situation générale.

Elles sont au nombre de trois :

- L'**initialisation** (notée Init) qui, lorsque la situation initiale est vérifiée, permet de passer à la situation générale.
- La **clôture** (notée Clôt) qui permet, lorsque l'invariant et la condition d'arrêt sont vérifiés, de passer à la situation finale.
- L'**itération** (notée Iter) qui lorsque l'invariant est vérifié et que la condition d'arrêt ne l'est pas, valide à nouveau l'invariant.

Plus formellement, soit S_1 et S_2 deux situations et I une suite d'instructions, la notation $\{S_1\}\ I\{S_2\}$ signifie :
'Si S_1 est vraie avant l'exécution de I, alors cette exécution se terminera et S_2 sera vérifiée."

Nous pouvons donc réécrire les trois points développés ci-dessus de la manière suivante :

- $\{S.I.\}$ Init $\{S.G.\}$
- $\{S.G.\ \wedge\ B\}$ Clôt $\{S.F.\}$
- $\{S.G.\ \wedge\ \neg B\}$ Iter $\{S.G.\}$

Ce que la figure [1.1] nous explique plus schématiquement.

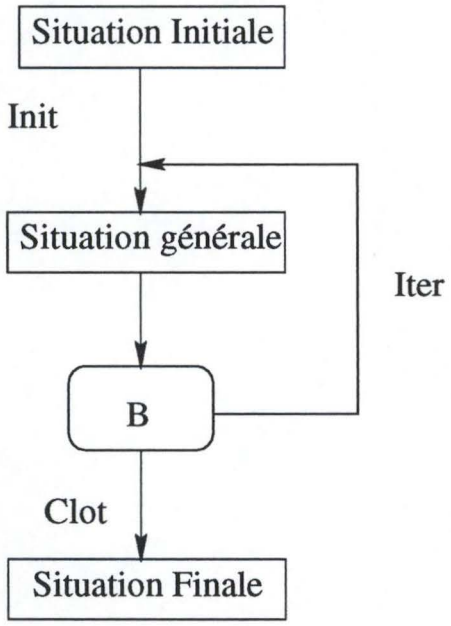


FIG. 1.1: Enchaînement des situations et instructions

1.2.4 Etape 4 : Vérification de la terminaison

Il faut vérifier qu'il existe une fonction entière et bornée des valeurs des variables, qui croît à chaque exécution de la suite d'instructions Iter.

Exemple : lorsqu'on utilise un indice croissant de manière strictement monotone pour parcourir un tableau, il suffit de prendre cet indice comme fonction.

1.2.5 Etape 5 : Ecriture

Les listes d'objets utilisés doivent être transformés en variables, constantes et tableaux et les suites d'instructions assemblées afin de former un programme Pascal.

1.3 Langage de description des schémas

Cette section contient une description des principaux concepts du **langage de description des schémas** (noté L.D.S). Le but de ce langage est de permettre à l'utilisateur de saisir les différentes situations sous forme graphique.

1.3.1 Le concept de schéma

Les situations sont décrites comme un ensemble de schémas. Un **schéma** étant une représentation des éléments d'un tableau et des conditions portant sur ceux-ci. Nous remarquons qu'il n'est pas interdit que plusieurs schémas représentent les éléments d'un même tableau sur lesquelles portent des conditions différentes.

1.3.2 Le concept de segment

Nous définissons un **segment** comme étant une suite d'éléments consécutifs d'un même tableau. Chaque schéma est représenté par une suite de segments contigus. Un segment possède donc une borne inférieure (resp. supérieure) qui est l'indice du premier (resp. dernier) élément de la suite. Il se caractérise par :

- la manière dont ses bornes sont fixées,

- les conditions portant sur son contenu,
- ses liens avec d'autres segments.

Propriétés d'un segment

- Un segment peut posséder un nom de référence.
- Un segment est vide si et seulement si sa borne inférieure est égale à sa borne supérieure plus un.
- Un segment est repéré par ses bornes.
  Celles-ci sont dites fixes lorsqu'elles sont liées à une variable, plus précisément quand la valeur de la borne est constamment égale à celle de la variable. Elle est mobile dans le cas contraire.
- Un segment est fixe ou mobile.
  Il est fixe si et seulement si ses deux bornes le sont. Il est mobile si et seulement si au moins une de ses bornes est mobile. Dans ce dernier cas, il représente l'ensemble des segments possibles dans l'espace de mobilité des bornes.

  Exemple : dans la figure (1.2) Les segments `X` et `Y` sont tous deux mobiles (ils ont chacun une borne mobile) tandis que le segment `Z` est fixe.

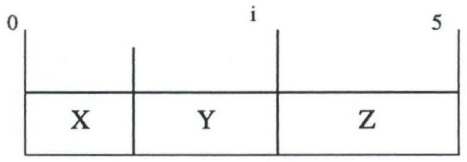


FIG. 1.2: Segments fixes et segments mobiles

Propriétés portant sur le contenu d'un segment

Les propriétés s'exprimeront par des conditions qui porteront sur la représentation graphique du segment. Ces conditions devront avoir une visualisation aussi intuitive que possible. Celles-ci pourront être des expressions

mathématiques mais comporteront des éléments plus puissants ou plus intuitifs.

Exemple :

`croissant`	signifie que les éléments du segment sont triés par ordre croissant
$\sum = s$	signifie que la somme des éléments du segment vaut s.
`permute`	signifie que les éléments du segment sont une permutation du contenu initial.

Liens entre segments

Pour pouvoir représenter au mieux les situations, il doit exister des mécanismes permettant de lier des segments entre eux. Par exemple pour représenter un tableau, il faut pouvoir exprimer le fait qu'il est constitué d'une suite de segments contigus. Nous distinguons les liens implicites des liens explicites.

Dans les liens implicites, nous retrouvons les relations qui peuvent s'exprimer par la présence des segments dans un même schéma.

- Deux segments situés dans un même schéma sont disjoints.
- Si deux segments sont contigus alors la borne supérieure du premier vaut la borne inférieure du second moins un.
- Si un segment est avant un autre alors sa borne supérieure est inférieure ou égale à la borne inférieure de l'autre moins un.

Les liens exprimés, eux, sont des liens qui ne sont pas directement visualisables dans un schéma. Ils sont par exemple de la forme :

$\sum = \sum(Y)$: la somme des éléments du segment est égale à celle des éléments du segment Y

1.3.3 Exemple de description d'une situation

Nous illustrons dans ce paragraphe, la manière de combiner des schémas pour parvenir à décrire une situation.

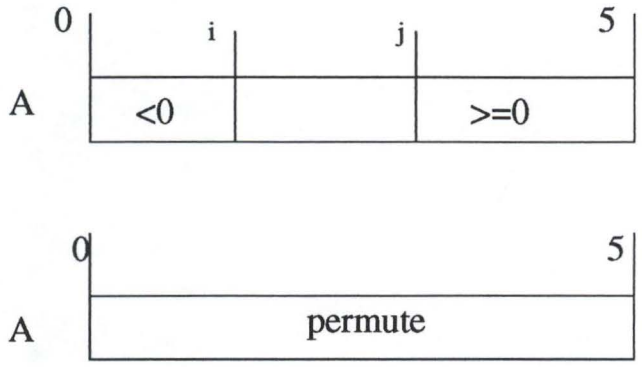


FIG. 1.3: Exemple de description de situation

Cette situation générale décrit le tableau A qui vérifie les conditions suivantes :

- le contenu de A est une permutation de son contenu initial,
- $A[1..i]$ est composé d'éléments strictement inférieurs à zéro,
- $A[j+1..5]$ est composé d'éléments supérieurs ou égaux à zéro.

Si nous analysons davantage le premier schéma, nous constatons que toutes les bornes (1, i, j, 5) sont fixes. Par conséquent, les segments le sont également. De plus, $A[1..i]$, $A[i+1..j]$, $A[j+1..5]$ sont contigus et disjoints.

1.4 Description de la méthode

Nous allons maintenant décrire les différentes étapes de la méthode

1.4.1 Spécifications

L'étudiant construit les spécifications du problème c'est-à-dire :

- introduire les objets utilisés,
- décrire la situation initiale,
- décrire la situation finale.

Le système vérifiera que toutes les variables utilisées pour décrire les situations ont bien été déclarées.

1.4.2 Situation générale et condition d'arrêt

Ensuite, il doit construire la situation générale. L'étudiant terminera cette étape en formulant la condition d'arrêt. Le système vérifiera que tous les objets ont bien été déclarés.

1.4.3 Choix des instructions

L'utilisateur doit maintenant introduire les différentes parties de code (Init, Iter, Clot).

Le programme vérifiera que toute variable apparaissant dans le code a déjà été déclarée. Il confirmera ou infirmera la conformité du code à la syntaxe Pascal. Enfin, il testera les relations du paragraphe (1.2.3).

Chapitre 2

Le cahier des charges

2.1 Spécification générale

Le système devra comporter les différents éléments suivants :

- Une fonctionnalité permettant la manipulation des objets ;
- Une fonctionnalité permettant la saisie des différentes situations (Initiale, Générale, Finale) ;
- Une fonctionnalité permettant la manipulation du code correspondant aux différentes suites d'instructions (Init, Iter,Clot) ;
- Une fonctionnalité permettant de saisir la condition d'arrêt et de la valider ;
- Une fonctionnalité vérifiant que l'enchaînement entre suites d'instructions et situations est correcte ;
- Une interface conviviale agençant les différentes fonctionnalités de manière ergonomique.

Les différents éléments seront spécifiés plus précisément dans les paragraphes qui suivent. L'interface, elle, ne sera pas détaillée dans ce chapitre, elle sera décrite plus en avant.

2.2 Spécification de la fonctionnalité de manipulation d'objets

Cette fonctionnalité devra permettre d'introduire des objets. Nous allons détailler chacun des types d'objets et voir ce qu'elle devra accomplir pour chacun d'entre eux.

Commençons par les tableaux d'entiers, la déclaration de ceux-ci devra s'effectuer de manière implicite au moment où l'utilisateur les dessine. Un tableau devra pouvoir être supprimé d'une ou de toutes les situations auxquelles il appartient. Cette suppression entraînera celle de tous les schémas associés à ce dernier. On devra pouvoir assigner des valeurs à tous les éléments d'un tableau ; ces valeurs devront être modifiables soit par l'utilisateur soit par le système.

La déclaration des variables requerra que l'utilisateur introduise un nom et un type pour chacune d'entre elles. Comme pour les tableaux, on pourra les supprimer et leur assigner une valeur. Elles pourront être modifiées par la suite soit par l'étudiant, soit par le programme.

Le contenu des tableaux et des variables devra être visualisable et la modification d'un objet entraînera une mise à jour automatique de l'affichage.

2.3 Spécification de la fonctionnalité graphique

L'interface graphique devra permettre de décrire une situation à partir de schémas. Pour ce faire, elle devra être capable de représenter tous les éléments du L.D.S. (voir paragraphe 1.3). Nous décrivons maintenant plus en avant cette interface.

Elle permettra de dessiner un tableau sous forme d'un rectangle[1]. Les arguments obligatoires à la création de cet objet seront

- un nom (unique) ;
- une borne inférieure (valeur numérique) ;
- une borne supérieure (valeur numérique).

L'utilisateur pourra introduire plus d'un schéma représentant le même tableau.

[1] Il s'agit en fait du premier schéma de ce tableau.

Il ajoutera éventuellement des bornes à chacun des schémas. Celles-ci seront soit mobiles, soit fixes. Dans le cas de bornes fixes, elles devront obligatoirement être associées à un nom de variable. Si cette variable n'est pas déclarée, alors le système l'ajoutera automatiquement. Leur représentation sera une barre verticale qui, dans le cas d'une borne fixe, sera accompagnée du nom de la variable qui lui est associée.

Ces bornes délimiteront des segments. Le caractère mobile ou fixe de ceux-ci sera déduit des propriétés des bornes les délimitant. L'utilisateur aura la possibilité d'ajouter les attributs falcultatifs suivants :

- un nom qui sera identifiant,
- une ou plusieurs conditions portant sur le contenu du segment.

Si un segment porte un nom, ce dernier sera indiqué dans sa représentation. De même s'il comporte une liste de conditions, sa présence sera signalée d'une manière quelconque.

L'utilisateur saisira les conditions à la manière d'un éditeur d'équations de style "Word". Les conditions comporteront, en plus des expressions logiques et arithmétiques traditionnelles les opérateurs exprimant les événements suivants :

- l'appartenance d'un élément au segment,
- le caractère trié ou non d'un segment,
- le contenu d'un segment est une permutation de son contenu initial,
- le contenu d'un segment équivaut à celui d'un autre,

et des fonctions telles que

- calcul du minimum et du maximum d'un segment,
- calcul de la somme, du produit des éléments d'un segment,
- calcul du nombre d'éléments d'un segment.

Dans une situation, chaque tableau pourra être supprimé, cette suppression entraînera la suppression de l'ensemble des schémas qui lui sont associés ; de plus si le tableau n'existe plus dans aucune situation, sa déclaration sera supprimée. La suppression d'un schéma entraînera la suppression de toutes les entités qui le composent. La suppression d'une borne ne pourra s'effectuer que dans le cas où elle délimite deux segments ; autrement dit, les bornes du tableau ne peuvent être retirées. Elle déclenchera la suppression de ces derniers et des conditions qui leur sont associées, et ensuite la création d'un segment regroupant les deux précités et ne comportant aucune condition.

Une représentation interne des situations sera construite à partir des graphiques. Cette représentation devra permettre de vérifier l'unicité des identifiants, l'existence de la variable liée à une borne fixe. Dans le cas d'une erreur, l'utilisateur en sera averti, et invité à la corriger. En plus, elle devra permettre de stocker les valeurs des éléments du tableau ainsi que tous les éléments nécessaires à l'exécution du programme.

2.4 Spécification de la fonctionnalité de manipulation de code

Cette fonctionnalité permettra de saisir les suites d'instructions (Init, Iter, Clot). Ces séquences se devront d'être conforme à la syntaxe du Pascal standard. Une fois encodées, celles-ci seront analysées par un parseur.

Lors de cette analyse, le parseur vérifiera que la syntaxe utilisée est correcte, sinon il en avisera l'utilisateur . Ensuite, il vérifiera que tous les objets apparaissant dans le code ont été déclarés. Si ce n'est pas le cas, il avertira l'utilisateur et lui demandera de déclarer ces objets. Il serait de plus souhaitable qu'il effectue une vérification de type sur les objets qu'il manipule et qu'en cas d'erreur, il le signale à l'utilisateur.

2.5 Spécification de la fonctionnalité de condition d'arrêt

Cette fonctionnalité permettra de saisir la condition d'arrêt. Cette condition aura la syntaxe d'une expression Pascal. Après analyse par un parseur, si le code est correct, cette expression sera stockée pour évaluation postérieure.

2.6 Fonctionnalité d'enchaînement

Le programme fera appel à cette fonctionnalité une fois que toutes les situations auront été décrites, les différentes séries d'instructions et la condition d'arrêt auront été validées.

Cette fonctionnalité aura pour but de vérifier que le code fourni et les situations décrites vérifient bien les relations décrites au paragraphe (1.2.3). Elle le fera en évaluant les situations, les expressions et en exécutant les

instructions. De plus, chaque fois qu'elle passera dans la boucle d'itération, elle l'indiquera.

Lorsqu'une situation ne sera pas vérifiée, le programme devra en avertir l'utilisateur aussi précisément que possible, en indiquant quelle situation n'est pas vérifiée et dans celle-ci, quelle condition ne l'est pas. Dans ce cas, l'exécution du programme sera stoppée et l'utilisateur pourra visualiser les variables et les tableaux pour mieux comprendre l'origine de son erreur. Dans tous les cas, il pourra alors réinitialiser tous les objets.

Remarquons qu'il serait aussi intéressant que cette fonctionnalité vérifie qu'un indice référençant un élément dans un tableau soit bien compris entre la borne inférieure et la borne supérieure de ce tableau.

Chapitre 3

Un langage de saisie des situations

L'outil d'aide à la programmation développé dans ce mémoire permet à l'utilisateur de décrire graphiquement les situations initiale, générale et finale. Il assure également une vérification syntaxique du programme Pascal introduit par l'utilisateur. La description des situations est utilisée lors de l'exécution du programme utilisateur afin de vérifier si les règles méthodologiques de la méthode de l'invariant ont été respectées. Celles-ci sont rappelées ci-dessous

- $\{S.I.\}$ Init $\{S.G.\}$
- $\{S.G. \wedge B\}$ Clôt $\{S.F.\}$
- $\{S.G. \wedge \neg B\}$ Iter $\{S.G.\}$

Les situations initiale, générale et finale sont désignées respectivement par S.I., S.G. et S.F. Le code de l'initialisation est représenté par Init, celui de l'itération par Iter et celui de clôture par Clôt. La condition d'arrêt est quant à elle représentée par B.

Pour pouvoir vérifier ces règles, il est donc nécessaire d'être en mesure de déterminer la validité des situations. Nous avons donc défini un langage graphique afin de disposer d'une représentation interne des situations et de pouvoir évaluer leur sémantique. A chaque situation, nous pouvons donc associer la valeur "vrai" ou "faux" selon les valeurs des objets du programme. Nous avons parcouru dans un premier temps les travaux [2], [6], [7] afin

d'étayer notre raisonnement. Nous présentons ci-dessous la syntaxe de notre langage graphique.

3.1 Syntaxe abstraite

Le but de cette syntaxe est de représenter une description d'assertions de programme ou, plus précisément, les préconditions et postconditions ainsi que la situation générale du programme.

Une description de programme est composée des situations initiale, générale et finale. Une situation décrit l'ensemble des tableaux et des variables y figurant et les conditions s'y rapportant.

$$
\begin{aligned}
< description > &::= < situation\ initiale >< situation\ générale > \\
&\quad\ < situation\ finale > \\
< situation\ initiale > &::= pre\ < tableaux >< variables >< conditions > \\
< situation\ générale > &::= sitgen\ < tableaux >< variables >< conditions > \\
< situation\ finale > &::= post\ < tableaux >< variables >< conditions >
\end{aligned}
$$

Un tableau est constitué d'une ou plusieurs représentations, chaque représentation étant partitionnée en un ou plusieurs segments. Des conditions facultatives sont définies sur ces segments.
Un segment est caractérisé par son nom et ses bornes qui peuvent être fixes ou mobiles.
Une borne fixe est liée à une variable définie par l'utilisateur, celle-ci référençant un indice du tableau. Une borne mobile n'est pas fixée par l'utilisateur.

$$
\begin{aligned}
< tableaux_1 > &::= vide \mid < tableau >< tableaux_0 > \\
< tableau > &::= < nom\,tableau >< b_i >< b_s >< suite\,représentations > \\
< suite\ représentations_1 > &::= vide
\end{aligned}
$$

$$
\begin{array}{rcl}
 & & | \ < \textit{représentation} >< \textit{suite représentations}_0 > \\
< \textit{représentation} > & ::= & < \textit{nom représentation} >< \textit{segments} > \\
< \textit{segments}_1 > & ::= & \textit{vide} \\
 & & | \ < \textit{segment} >< \textit{segments}_0 > \\
< \textit{segment} > & ::= & < \textit{nom segment} > \ \{f/m\} < \textit{borne}_g > \\
 & & \{f/m\} < \textit{borne}_d > \\
 & & < \textit{conditions} > \\
< \textit{conditions}_1 > & ::= & \textit{vide} \\
 & & | < \textit{condition} >< \textit{conditions}_0 > \\
< \textit{condition} > & ::= & < \textit{expr} - \textit{bool} > \\
< \textit{variables}_1 > & ::= & \textit{vide} \mid \ < \textit{variable} >< \textit{variables}_0 > \\
< \textit{variable} > & ::= & \textit{var} \ < \textit{identificateur} > : < \textit{type} > \\
< \textit{type} > & ::= & \textit{booleen} \mid \textit{entier} \\
< \textit{nom tableau} > & ::= & < \textit{identificateur} > \\
< b_i > & ::= & < \textit{identificateur} > \\
 & & | < \textit{constante} > \\
< b_s > & ::= & < \textit{identificateur} > \\
 & & | < \textit{constante} > \\
< \textit{nom représentation} > & ::= & < \textit{identificateur} > \\
< \textit{nom segment} > & ::= & < \textit{identificateur} > \\
< \textit{borne}_g > & ::= & < \textit{identificateur} > \\
 & & | < \textit{constante} > \\
< \textit{borne}_d > & ::= & < \textit{identificateur} > \\
 & & | < \textit{constante} > \\
< \textit{identificateur} > & ::= & < \textit{lettre} > \{< \textit{lettre} > | < \textit{chiffre} >\}^* \\
< \textit{constante} > & ::= & \{1|2|3|4|5|6|7|8|9\}\{< \textit{chiffre} >\}^* \\
< \textit{lettre} > & ::= & a|..|z|A|..|Z \\
< \textit{chiffre} > & ::= & 0|1|2|3|4|5|6|7|8|9
\end{array}
$$

Les fonctions booléennes sont des fonctions se rapportant à un segment et fournissant un résultat booléen. Celles-ci peuvent faire intervenir des fonc-

tions entières.

$$
\begin{aligned}
< fonct - bool > \quad ::= \quad & appartient < expr - ent > \\
& | \ tricroiss \\
& | \ tristrcroiss \\
& | \ tridecroiss \\
& | \ tristrdecroiss \\
& | \ permute(< nom\ segment >) \\
& | \ inchangé(< nom\ segment >) \\
& | \ egal \\
< fonct - ent > \quad ::= \quad & somme \\
& | \ produit \\
& | \ min \\
& | \ max \\
& | \ longueur \\
< expr - ent_2 > \quad ::= \quad & < expr - ent_1 >< op - ent >< expr - ent_0 > \\
& | \ (< expr - ent_1 >) \\
& | \ < constante > \\
& | \ < identificateur > \\
& | \ < identificateur > [< expr - ent_1 >] \\
& | \ < fonct - ent > \\
& | \ moinsu < expr - ent_1 > \\
< expr - bool_2 > \quad ::= \quad & < expr - bool_1 >< op - bool >< expr - bool_0 > \\
& | \ (< expr - bool_1 >) \\
& | \ < const - bool > \\
& | \ < identificateur > \\
& | \ < fonct - bool > \\
& | \ < expr - ent_1 >< op - rel >< expr - ent_0 > \\
& | \ pourtout < identificateur >:< expr - bool_1 > \\
& | \ ilexiste < identificateur >:< expr - bool_1 >
\end{aligned}
$$

$$
\begin{array}{rcl}
 & | & non < expr - bool_1 > \\
< const - bool > & ::= & vrai \\
 & | & faux \\
< op - rel > & ::= & egal \\
 & | & diff \\
 & | & pp \\
 & | & ppe \\
 & | & pg \\
 & | & pge \\
< op - ent > & ::= & plus \\
 & | & moins \\
 & | & fois \\
 & | & div
\end{array}
$$

3.2 L'environnement et le store

L'**environnement** e d'une description de programme est une fonction des identificateurs dans les *left-values* :

$$e : Id \rightarrow L_V$$

L'ensemble des *left-values* est défini par :

$$L_V = L + (\mathbb{N} \rightarrow L) + \{undef\}$$

où L est l'ensemble des locations mémoires et $\mathbb{N}$ celui des entiers naturels.

Le **store** s est une fonction des locations vers les *right-values* :

$$s : L \rightarrow R_V$$

où l'ensemble des right-values est défini par

$$R_V = \mathbb{Z} + \mathbb{B}$$

où $\mathbb{Z}$ est l'ensemble des entiers et $\mathbb{B}$ l'ensemble des booléens.

Par abus de langage, le terme environnement initial désignera l'environnement et le store initiaux, c'est-à dire ceux définis avant la situation initiale.

3.3 La sémantique

Par la suite, e et s désigneront respectivement un environnement et un store quelconques, tandis que e_0 et s_0 désigneront l'environnement et le store initiaux. A chaque segment est associée une fonction f permettant d'accéder aux éléments du segment. Autrement dit, la fonction f de domaine $\mathbb{N}$ à valeur dans $\mathbb{Z} \cup \{noval\}$ associe à un indice la valeur stockée dans la cellule du segment désignée par l'indice.

Plus précisément, supposons que la fonction entière s'applique à un segment de nom $< nom\ segment >$. Nous notons f, la fonction qui associe à chaque indice i du segment, la valeur $s[i]$ et nous posons $f = Valseg[\![< nom\ segment > < borne_g > < borne_d >]\!]\ e\ s$ (cfr plus loin).

Une fonction sémantique sera associée à chaque objet syntaxique.

3.3.1 Sémantique d'une expression entière

La sémantique d'une expression entière est donnée par la fonction $\mathcal{E}$. Elle a pour signature :

$$\{< expr - ent >\} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z} \cup \{noval\}) \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{Z}.$$

Remarquons qu'un objet syntaxique tel que $< expr - ent >$ encadré d'accolades représente l'ensemble des valeurs que peut prendre cet objet. De plus, l'expression $(\mathbb{N} \rightarrow \mathbb{Z} \cup \{noval\})$ désigne l'ensemble des fonctions f décrites plus haut. Aussi, les expressions $\mathbb{E}$ et $\mathbb{S}$ sont respectivement l'ensemble des environnements et des stores.

Pour une expression entière comportant un opérateur binaire : $< expr - ent_1 >< op - ent >< expr - ent_0 >$, la sémantique est obtenue en évaluant la sémantique des deux expressions et en appliquant l'opérateur binaire associé à $< op - ent >$.

$$\mathcal{E}[\![< expr - ent_1 >< op - ent >< expr - ent_0 >]\!]\ f\ e\ s = (\mathcal{E}[\![< expr - ent_1 >]\!]\ f\ e\ s)\ \mathcal{O}[\![< op - ent >]\!]\ (\mathcal{E}[\![< expr - ent_0 >]\!]\ f\ e\ s)$$

La sémantique d'une expression entière parenthésée est la sémantique de l'expression entière non parenthésée.

$$\mathcal{E}[\![(< expr - ent_1 >)]\!]\ f\ e\ s\ =$$
$$\mathcal{E}[\![< expr - ent_1 >]\!]\ f\ e\ s$$

La sémantique d'une constante est cette constante.

$$\mathcal{E}[\![< constante >]\!]\ f\ e\ s\ =\ < constante >$$

La sémantique d'un identificateur est la valeur qui lui est associée pour l'environnement et le store donnés. Nous distinguons le cas des variables simples :

$$\mathcal{E}[\![< identificateur >]\!]\ f\ e\ s\ =$$
$$s(e(< identificateur >))$$

de celui des variables indicées :

$$\mathcal{E}[\![< identificateur > [< expr - ent_1 >]]\!]\ f\ e\ s\ =$$
$$e(< identificateur >)(\mathcal{E}(< expr - ent_1 >)\ f\ e\ s)$$

La sémantique d'une fonction entière nous est donnée par la fonction $\mathcal{F}e$ décrite ci-dessous.

$$\mathcal{E}[\![< fonct - ent >]\!]\ f\ e\ s\ =\ \mathcal{F}e[\![< fonct - ent >]\!]\ f\ e\ s$$

La sémantique de l'opposé d'une expression entière est l'opposé de sa sémantique.

$$\mathcal{E}[\![moinsu < expr - ent_1 >]\!]\ f\ e\ s\ =$$
$$-\mathcal{E}[\![< expr - ent_1 >]\!]\ f\ e\ s$$

3.3.2 Sémantique d'une fonction entière

La sémantique d'une fonction entière est donnée par la fonction $\mathcal{F}e$. Sa signature est :

$$\{< fonct - ent >\} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z} \cup \{noval\}) \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{Z}.$$

La sémantique de la fonction somme est la somme des éléments du segment sur lequel elle porte. Ces éléments sont obtenus grâce à la fonction f.

$$\mathcal{F}e[\![somme]\!]\ f\ e\ s\ =\ \sum_{dom(f)} f(i)$$

La sémantique de la fonction produit est le produit des éléments du segment sur lequel elle porte.

$$\mathcal{F}e[\![produit]\!]\ f\ e\ s\ =\ \prod_{dom(f)} f(i)$$

La sémantique de la fonction min est la valeur minimale des éléments du segment sur lequel elle porte.

$$\mathcal{F}e[\![min]\!]\ f\ e\ s\ =\ \min_{dom(f)} f(i)$$

La sémantique de la fonction max est la valeur maximale des éléments du segment sur lequel elle porte.

$$\mathcal{F}e[\![max]\!]\ f\ e\ s\ =\ \max_{dom(f)} f(i)$$

La sémantique de la fonction longueur est la longueur du segment i.e. son nombre d'éléments.

$$\mathcal{F}e[\![longueur]\!]\ f\ e\ s\ =\ \#dom(f)$$

3.3.3 Sémantique d'une expression booléenne

La sémantique d'une expression booléenne est donnée par la fonction $\mathcal{B}$. Sa signature est :

$$\{< expr - bool >\} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z} \cup \{noval\}) \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}.$$

Pour une expression booléenne comportant un opérateur binaire : $< expr - bool_1 >< op - bool >< expr - bool_0 >$, la sémantique est obtenue en évaluant la sémantique des 2 expressions et en appliquant l'opérateur binaire associé à $< op - bool >$.

$$\mathcal{B}[\![< expr - bool_1 >< op - bool >< expr - bool_0 >]\!]\ f\ e\ s = (\mathcal{B}[\![< expr - bool_1 >]\!]\ f\ e\ s)\ \mathcal{O}[\![< op - bool >]\!]\ (\mathcal{B}[\![< expr - bool_0 >]\!]\ f\ e\ s)$$

La sémantique d'une expression booléenne parenthésée est la sémantique de l'expression booléenne non parenthésée.

$$\mathcal{B}[\![(< expr - bool_1 >)]\!]\ f\ e\ s\ =\ \mathcal{B}[\![< expr - bool_1 >]\!]\ f\ e\ s$$

La sémantique d'une constante booléenne est fournie par la fonction $\mathcal{C}b$.

$$\mathcal{B}[\![< const - bool >]\!]\ f\ e\ s\ =\ \mathcal{C}b < const - bool >$$

La sémantique d'une variable booléenne est la valeur qui lui est associée dans l'environnement et le store donnés.

$$\mathcal{B}[\![< identificateur >]\!]\ f\ e\ s\ =\ s(e(< identificateur >))$$

La sémantique d'une fonction booléenne nous est donnée par la fonction $\mathcal{F}b$ décrite ci-dessous.

$$\mathcal{B}[\![< fonct - bool >]\!]\ f\ e\ s\ =\ \mathcal{F}b[\![< fonct - bool >]\!]\ f\ e\ s$$

Pour une expression booléenne comportant un opérateur relationnel : $< expr - bool_1 >< op - rel >< expr - bool_0 >$, la sémantique est obtenue en évaluant la sémantique des 2 expressions et en appliquant l'opérateur relationnel associé à $< op - rel >$.

$$\mathcal{B}[\![< expr - ent_1 >< op - rel >< expr - ent_0 >]\!]\ f\ e\ s\ =$$
$$(\mathcal{E}[\![< expr - ent_1 >]\!]\ f\ e\ s)\ \mathcal{O}r[\![< op - rel >]\!]\ (\mathcal{E}[\![< expr - ent_0 >]\!]\ f\ e\ s)$$

La sémantique de la négation d'une expression booléenne est la négation de la sémantique de cette expression.

$$\mathcal{B}[\![non < expr - bool_1 >]\!]\ f\ e\ s\ =\ not\ (\mathcal{B}[\![< expr - bool_1 >]\!]\ f\ e\ s)$$

La sémantique d'une expression du type "$pourtout\ < identificateur > : < expr - bool >$" a pour but de tester si chaque élément du segment vérifie la condition :

$$\mathcal{B}[\![pourtout\ < identificateur > : < expr - bool_1 >]\!]\ f\ e\ s\ =$$
$$\forall x \in codom(f)\ \mathcal{B}[\![< expr - bool_1 >]\!]\ f\ e\ s$$

La sémantique d'une expression du type "$ilexiste\ < identificateur > : < expr - bool >$" a pour but de tester si un élément du segment vérifie la condition :

$$\mathcal{B}[\![ilexiste\ < identificateur > : < expr - bool_1 >]\!]\ f\ e\ s\ =$$
$$\exists x \in codom(f)\ \mathcal{B}[\![< expr - bool_1 >]\!]\ f\ e\ s$$

L'expression $< expr - bool_1 >$ doit contenir la variable que l'objet syntaxique $< identificateur >$ désigne.

3.3.4 Sémantique d'une constante booléenne

La sémantique d'une constante booléenne nous est fournie par la fonction $\mathcal{C}b$ décrite dans le tableau suivant :

$< const - bool >$	$\mathcal{C}b$
$vrai$	$true$
$faux$	$false$

3.3.5 Sémantique d'un opérateur relationnel

La sémantique d'un opérateur relationnel nous est fourni par le tableau suivant qui décrit la fonction sémantique $\mathcal{O}r$:

$< op - rel >$	$\mathcal{O}r$
$egal$	$=$
$diff$	$\neq$
pp	$<$
ppe	$\leq$
pg	$>$
pge	$\geq$

3.3.6 Sémantique d'une fonction booléenne

La sémantique d'une fonction booléenne est donnée par la fonction $\mathcal{F}b$ dont la signature est :

$$\{< fonct - bool >\} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z} \cup \{noval\}) \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

Pour la suite, nous définissons $b_i = \mathcal{E}[\![< borne_g >]\!]\ e\ s$ et $b_s = \mathcal{E}[\![< borne_d >]\!]\ e\ s$.

La sémantique d'une fonction booléenne de la forme

$$appartient\ \ < expr - ent >$$

est donnée par :

$$\mathcal{F}b[\![appartient\ \ < expr - ent >]\!]\ f\ e\ s\ \ =\ \ (\mathcal{E}[\![< expr - ent >]\!]\ f\ e\ s) \in codom(f)$$

La sémantique d'une fonction booléenne de la forme *tricroiss* est donnée par :

$$\mathcal{F}b[\![tricroiss]\!]\ f\ e\ s\ =\ \forall i\ :\ b_i \leq i < b_s\ :\ f(i) \leq f(i+1)$$

La sémantique d'une fonction booléenne de la forme *tristrcroiss* est donnée par :

$$\mathcal{F}b[\![tristrcroiss]\!]\ f\ e\ s\ =\ \forall i\ :\ b_i \leq i < b_s\ :\ f(i) < f(i+1)$$

La sémantique d'une fonction booléenne de la forme *tridecr* est donnée par :

$$\mathcal{F}b[\![tridecr]\!]\ f\ e\ s\ =\ \forall i\ :\ b_i \leq i < b_s\ :\ f(i) \geq f(i+1)$$

La sémantique d'une fonction booléenne de la forme *tristrdecr* est donnée par :

$$\mathcal{F}b[\![tristrdecr]\!]\ f\ e\ s\ =\ \forall i\ :\ b_i \leq i < b_s\ :\ f(i) > f(i+1)$$

La sémantique d'une fonction booléenne de la forme $permute(X)$ est donnée par :

$$\mathcal{F}b[\![permute(X)]\!]\ f\ e\ s\ =\ true$$

ssi le multi-ensemble déterminé par l'image de f est égal à celui déterminé par l'image de f' où f' représente les valeurs de X dans l'environnement et le store actuels i.e.

$$f' = Valseg[\![X, b_i, b_s]\!]\ e\ s.$$

La sémantique d'une fonction booléenne de la forme *inchangé*(X) est donné par :

$$\mathcal{F}b[\![inchangé(X)]\!]\ f\ e\ s\ =\ \forall i\ :\ b_i \leq i \leq b_s\ f(i) = f_0(i)$$

où f_0 représente les valeurs de X dans l'environnement et le store initiaux i.e.

$$f_0 = Valseg[\![X, b_i, b_s]\!]\ e_0\ s_0.$$

Signalons que, dans ce dernier cas, l'argument X a été introduit pour faciliter l'expression de la sémantique ; logiquement, X devrait toujours porter le nom du segment auquel la condition se rapporte. Cet argument n'existera dès lors plus dans la syntaxe concrète.

La sémantique d'une fonction booléenne de la forme *egal* est donnée par :

$$\mathcal{F}b[\![egal]\!]\ f\ e\ s\ =\ \forall i\ :\ b_i \leq i < b_s\ :\ f(i) = f(i+1)$$

3.3.7 Sémantique d'une suite de conditions

La sémantique d'une suite de conditions est donnée par la fonction $\mathcal{C}s$. Sa signature est :

$$\{< conditions >\} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z} \cup \{noval\}) \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}.$$

La sémantique d'une suite de conditions de la forme *vide* est donnée par :

$$\mathcal{C}s[\![vide]\!]\ f\ e\ s\ =\ true$$

La sémantique d'une suite de conditions de la forme
$< condition >< conditions >$ est donnée par :

$$\begin{aligned}\mathcal{C}s[\![< condition > < conditions_0 >]\!]\ f\ e\ s\ =\ & (\mathcal{C}[\![< condition >]\!]\ f\ e\ s) \\ & \wedge(\mathcal{C}s[\![< conditions_0 >]\!]\ f\ e\ s)\end{aligned}$$

3.3.8 Sémantique d'une condition

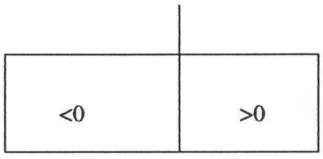
La sémantique d'une condition est donnée par la fonction $\mathcal{C}$. Sa signature est :

$$\{< condition >\} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z} \cup \{noval\}) \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}.$$

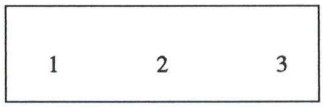
$$\begin{aligned} \mathcal{C}[\![vide]\!]\ f\ e\ s &= true \\ \mathcal{C}[\![< condition >]\!]\ f\ e\ s &= (\mathcal{B}[\![< expr - bool >]\!]\ f\ e\ s) \wedge (dom(f) \neq \emptyset) \end{aligned}$$

Ceci implique qu'un segment sans condition prendra toujours une valeur vraie tandis qu'un segment vide avec conditions prendra toujours une valeur fausse. Il nous semble que cela traduit au mieux la situation définie par l'utilisateur. Pour le voir, considérons l'exemple suivant :

Si l'utilisateur introduit une situation de ce type,



et si les segments vides sont toujours interprétés comme vrais, alors la situation suivante serait interprétée comme vraie,



ce qui ne nous semble pas intuitif.

De plus, dans ce cas, les bornes mobiles ne seraient pas d'un grand intérêt puisqu'il suffirait de réduire un segment mobile au vide pour qu'il prenne le sens "vrai".

Par contre, dans le cas de notre sémantique i.e. s'il y a une condition, le segment doit être non vide pour que ce soit vrai, le cas ci-dessus prend la valeur "faux".

Nous sommes conscients qu'il s'agit d'un choix et qu'il existe des exemples pour lesquels l'interprétation choisie poserait problème. Nous voulons dire

que dans certains cas, la sémantique d'une condition portant sur un segment vide devrait être vraie. Ce qui est le cas de la recherche dichotomique dont nous reprenons ci-dessous la situation générale (cfr figure 3.1).

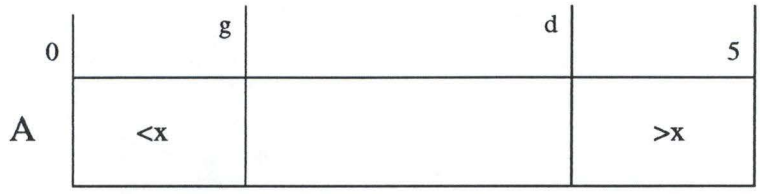


FIG. 3.1: Recherche dichotomique de x

En effet, cette situation doit être vérifiée avec les bornes fixes g et d initialisées respectivement à 0 et 5, les segments délimités par ces bornes et celles du tableau sont alors réduits au vides.

C'est pourquoi, lors de l'implémentation, nous avons remanié légèrement la sémantique en distinguant les segments fixes des segments mobiles. Pour les segments mobiles, nous gardons la sémantique décrite ci-dessus, tandis que pour les segments fixes, nous appliquons les règles suivantes :

$$\begin{aligned} \mathcal{C}[\![vide]\!]\ f\ e\ s &= true \\ \mathcal{C}[\![< condition >]\!]\ f\ e\ s &= \mathcal{B}[\![< expr - bool >]\!]\ f\ e\ s \end{aligned}$$

3.3.9 Sémantique d'un segment

La sémantique d'un segment est évaluée de la façon suivante : dans un premier temps, nous calculons la fonction f associée au segment grâce à la fonction sémantique $Valseg$; ensuite, nous vérifions la validité des conditions à partir de f et grâce à $\mathcal{C}$.

Nous définissons d'abord $Valseg$. Cette fonction associe à chaque indice du segment la valeur qui y correspond. Aux entiers n'appartenant pas à l'ensemble des indices, cette fonction leur associe *noval*.

Sa signature est donnée par :

$$\{< segment >\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z} \cup \{noval\}).$$

Elle est décrite par

$$Valseg[\![< nom\ segment >< borne_g >< borne_d >]\!]\ e\ s =$$
$$\lambda_i.cond(\mathcal{E}[\![< borne_g >]\!]\ e\ s < i \leq \mathcal{E}[\![< borne_d >]\!]\ e\ s,$$
$$\mathcal{E}[\![< nom\ segment > [i]]\!]\ e\ s,$$
$$noval)$$

Nous imposons que les indices identifiant les bornes sont situés à gauche de celles-ci, ce qui explique la stricte inégalité située avant le i.

A partir de cette fonction, nous pouvons définir la sémantique d'un segment. Celle-ci nous est donnée par la fonction $\mathcal{S}eg$ dont la signature est :

$$\{< segment >\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}.$$

Elle est décrite par :

$$\mathcal{S}eg[\![< nom\ segment >\ \{f/m\} < borne_g >\ \{f/m\} < borne_d >]\!]\ e\ s =$$
$$\mathcal{C}s[\![< conditions >]\!]\ (Valseg[\![< nom\ segment >< borne_g >< borne_d >]\!]\ e\ s)\ e\ s)$$

3.3.10 Sémantique d'un bloc

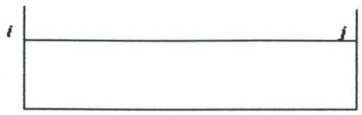
La notion de bloc a été introduite pour tenir compte des bornes mobiles et fixes. Un bloc est une suite de segments ayant une des trois configurations suivantes :

1. un segment fixe (appelé insécable et désigné par I)
2. deux segments mobiles (appelé sécable et désigné par S_1)
3. plusieurs segments mobiles (appelé sécable et désigné par S_p)

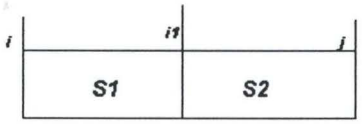
Nous donnons successivement la sémantique associée à ces trois cas :

Premier cas La sémantique d'un bloc insécable (I) est celle du segment fixe (S) qu'il représente.

$$\mathcal{B}l[\![I]\!]\ e\ s = \mathcal{S}eg[\![S]\!]\ e\ s$$



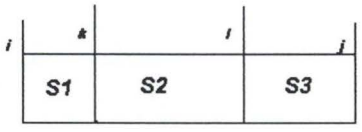
Deuxième cas Pour décrire la sémantique du deuxième cas, nous utilisons le dessin suivant :



Celle-ci est donnée par :

$$\mathcal{B}l[\![S_1]\!]\ e\ s\ =\ \exists i_1 \dagger i \leq i_1 \leq j \dagger \mathcal{S}eg[\![s_1]\!]\ e\ s\ \wedge \mathcal{S}eg[\![s_2]\!]\ e\ s$$

Troisième cas Pour décrire la sémantique du troisième cas, nous utilisons le dessin suivant :



Celle-ci est donnée par :

$$\mathcal{B}l[\![S_p]\!]\ e\ s\ =\ \exists\, l_1 \dagger i \leq l_1 \leq j \dagger \mathcal{S}eg[\![s_1]\!]\ e\ s$$

$$
\begin{array}{l}
\wedge\ \exists\ l_2 \dagger l_1 \leq l_2 \leq j \dagger \mathcal{S}eg[\![s_2]\!]\ e\ s \\
\vdots \\
\wedge\ \exists\ l_n \dagger l_{n-1} \leq l_n \leq j \dagger \mathcal{S}eg[\![s_n]\!]\ e\ s \\
\wedge\ \mathcal{S}eg[\![s_{n+1}]\!]\ e\ s
\end{array}
$$

Pour les objets syntaxiques suivants, nous conseillons au lecteur de retourner à la section 3.1 décrivant la syntaxe abstraite afin qu'il se remémore comment s'enchaînent les autres notions de la syntaxe. Pour rappel, une situation est composée d'une suite de tableaux. Un tableau se divise en une suite de représentations et une représentation est constituée d'une suite de segments.

3.3.11 Sémantique d'une représentation

La sémantique d'une représentation est la conjonction de la sémantique des blocs qui la composent. Elle est donnée par la fonction $\mathcal{R}$ et a pour signature :

$$\{< representation >\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

Comme une représentation est constituée d'une suite de blocs, nous posons afin de définir son sens

$$< representation >= (< bloc >_i)_{1 \leq i \leq n}.$$

La fonction $\mathcal{R}$ est définie par :

$$
\begin{array}{rcl}
\mathcal{R}[\![< representation >]\!]\ e\ s & = & (\mathcal{B}l[\![< bloc >_1]\!]\ e\ s) \\
& & \wedge ... \\
& & \wedge\ (\mathcal{B}l[\![< bloc >_n]\!]\ e\ s)
\end{array}
$$

3.3.12 Sémantique d'une suite de représentations

La sémantique d'une suite de représentations nous est donnée par la fonction $\mathcal{R}s$ dont la signature est :

$$\{< suite\ représentations >\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

La sémantique d'une suite de représentations de forme $< vide >$ nous est donnée par :

$$\mathcal{R}s[\![< vide >]\!]\ e\ s\ =\ true$$

La sémantique d'une suite de représentations de forme
$< \textit{représentation} >< \textit{suite représentations} >$ nous est donnée par :

$$\mathcal{R}s[\![< \textit{représentation} >< \textit{suite représentations} >]\!]\ e\ s$$
$$= \mathcal{R}[\![< \textit{représentation} >]\!]\ e\ s \wedge \mathcal{R}s[\![< \textit{suite représentations}]\!]\ e\ s$$

3.3.13 Sémantique d'un tableau

La sémantique d'un tableau nous est donnée par la fonction $\mathcal{T}$. Celle-ci correspond à la sémantique de sa suite de représentations. La signature de $\mathcal{T}$ est donnée par :

$$\{< \textit{tableau} >\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

$$\mathcal{T}[\![< \textit{tableau} >]\!]\ e\ s = \mathcal{R}s[\![< \textit{suite représentations} >]\!]\ e\ s$$

3.3.14 Sémantique d'une suite de tableaux

La sémantique d'une suite de tableaux nous est donnée par la fonction $\mathcal{T}s$. Celle-ci calcule la conjonction des sémantiques des différents tableaux. Sa signature est :

$$\{< \textit{tableaux} >\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

Si $< \textit{tableaux} >= \textit{vide}$, alors

$$\mathcal{T}s[\![\textit{vide}]\!]\ e\ s = \textit{true}$$

Si $< \textit{tableaux} >$ est de la forme $< \textit{tableau} >< \textit{tableaux}_0 >$, alors

$$\mathcal{T}s[\![< \textit{tableau} >< \textit{tableaux}_0 >]\!]\ e\ s \ = \ \mathcal{T}[\![< \textit{tableau} >]\!]\ e\ s \wedge \mathcal{T}s[\![< \textit{tableaux}_0 >]\!]\ e\ s$$

3.3.15 Sémantique d'une situation

La sémantique d'une situation est la sémantique de la suite de tableaux qui la compose i.e.

$$\mathcal{S}[\![< \textit{situation} >]\!]\ e\ s = \mathcal{T}s[\![< \textit{tableaux} >]\!]\ e\ s$$

Le mot-clé *pre* signifie que la situation doit être vraie au lancement du programme (avant toute instruction).
Le mot-clé *sitgen* signifie que la situation doit être vraie à chaque itération, il est sous-entendu qu'une situation générale se rapporte à une boucle.
Le mot-clé *post* signifie que la situation doit être vraie après l'exécution du programme.

3.3.16 Sémantique d'une description

Il s'agit de l'adéquation entre le programme et sa description. Pour ce faire, nous vérifions qu'au lancement du programme la situation initiale est vérifiée, que la situation générale est satisfaite à chaque itération et que la situation finale est également vérifiée en fin de programme.

3.3.17 Limitations

La syntaxe abstraite telle que définie ne nous permet pas d'exprimer des conditions faisant intervenir des fonctions entières portant sur des segments différents. Par exemple, on ne peut pas exprimer la condition $\sum = \sum(Y)$ qui signifie que la somme des éléments du segment portant la condition est égale à celle des éléments de Y.

Pour formuler de telles conditions, il est nécessaire d'introduire des variables intermédiaires. Nous verrons à l'aide d'un exemple comment procéder dans le chapitre 5.

3.4 Syntaxe concrète

La syntaxe concrète de notre langage graphique diffère de la syntaxe abstraite uniquement au niveau des expressions booléennes et entières. C'est pourquoi, nous ne détaillons ci-dessous que la partie de la syntaxe concrète relative aux expressions. L'intérêt d'un tel développement est qu'il prend en compte les niveaux de priorité des différents opérateurs.

3.4.1 Description de la syntaxe

La syntaxe des expressions booléennes est donnée par :

$$
\begin{array}{rcl}
< \textit{expression booléenne} > & ::= & < \textit{expression booléenne} > \ \textit{or} \ < \textit{terme booléen} > \\
& | & < \textit{terme booléen} > \\
< \textit{terme booléen} > & ::= & < \textit{terme booléen} > \ \textit{and} \ < \textit{facteur booléen} > \\
& | & < \textit{facteur booléen} > \\
< \textit{facteur booléen} > & ::= & < \textit{const} - \textit{booléenne} > \\
& | & < \textit{identificateur} > \\
& | & < \textit{fonct} - \textit{booléenne} > \\
& | & \textit{not} \ < \textit{facteur booléen} > \\
& | & (< \textit{expression booléenne} >) \\
& | & < \textit{expression arithmétique} >< \textit{op. relationnel} > \\
& & < \textit{expression arithmétique} > \\
& | & \textit{pourtout} \ < \textit{identificateur} > : < \textit{expression booléenne} > \\
& | & \textit{ilexiste} \ < \textit{identificateur} > : < \textit{expression booléenne} > \\
< \textit{const} - \textit{booléenne} > & ::= & \textit{true} \\
& | & \textit{false} \\
< \textit{identificateur} > & ::= & < \textit{lettre} > \{< \textit{lettre} > | < \textit{chiffre} >\}^* \\
< \textit{lettre} > & ::= & a|..|z|A|..|Z \\
< \textit{chiffre} > & ::= & 0|1|2|3|4|5|6|7|8|9 \\
< \textit{fonct} - \textit{booléenne} > & ::= & \textit{appartient} < \textit{expression arithmétique} > \\
& | & \textit{tricroiss} \\
& | & \textit{tristrcroiss} \\
& | & \textit{tridecroiss} \\
& | & \textit{tristrdecroiss} \\
& | & \textit{permute} < \textit{identificateur} > \\
& | & \textit{inchangé} \\
& | & \textit{egal} \\
< \textit{nom segment} > & ::= & < \textit{identificateur} >
\end{array}
$$

$$< op.\ relationnel > \quad ::= \quad = \mid <> \mid < \mid <= \mid > \mid >=$$

Les expressions arithmétiques ont pour syntaxe :

$$\begin{array}{rcl}
< expression\ arithmétique > & ::= & < expression\ arithmétique >< op.\ additif > \\
 & & < terme\ arithmétique > \\
 & \mid & < terme\ arithmétique > \\
< terme\ arithmétique > & ::= & < terme\ arithmétique >< op.\ multiplicatif > \\
 & & < facteur\ arithmétique > \\
 & \mid & < facteur\ arithmétique > \\
< facteur\ arithmétique > & ::= & < const - entière > \\
 & \mid & < identificateur > \\
 & \mid & < identificateur > \ [< expression\ arithmétique >] \\
 & \mid & < fonct - entière > \\
 & \mid & - \ < facteur\ arithmétique > \\
 & \mid & (< expression\ arithmétique >) \\
< op.\ additif > & ::= & + \mid - \\
< op.\ multiplicatif > & ::= & * \mid / \\
< const - entière > & ::= & \{1|2|3|4|5|6|7|8|9\}\{< chiffre >\}^* \\
< fonct - entière > & ::= & somme \\
 & \mid & produit \\
 & \mid & min \\
 & \mid & max \\
 & \mid & longueur
\end{array}$$

3.4.2 Exemples d'utilisation

Dans cette section, nous illustrons la syntaxe concrète par quelques exemples d'utilisation. Nous donnerons en fait uniquement des exemples se rapportant à la syntaxe des expressions ; la seule partie de la syntaxe devant être manipulée par l'utilisateur de l'outil d'aide à la programmation développé.

Dans les exemples qui suivent, et c'est aussi le cas dans l'outil d'aide à la programmation, les expressions sont schématisées par un cercle. Le dessin ci-dessous signifie qu'une condition (une expression) est associée au tableau A.

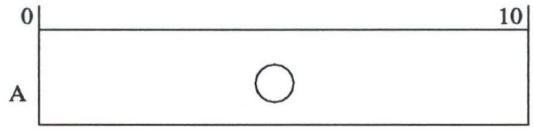


FIG. 3.2: Condition dans un tableau

Fonctions entières dans une expression

La condition "*somme* $= 3$" dans le tableau de la figure 3.3 ci-dessous signifie que la somme des éléments $A[3], \ldots, A[7]$ vaut 3.

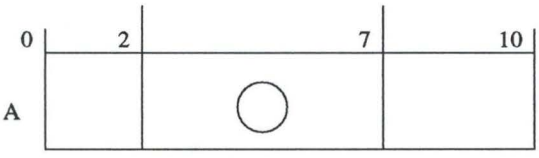


FIG. 3.3: Condition dans un segment

Le terme de droite aurait bien sûr pu être remplacé par une expression plus compliquée : par exemple, $(9 - 2 * 3)/(2/2)$.

La condition "*somme* $=$ *produit*" dans le même tableau de la figure 3.3 signifie que

$$\sum_{i=3}^{7} A[i] = \prod_{i=3}^{7} A[i]$$

Une condition peut également porter sur l'entièreté d'un tableau qui dans ce cas, ne comporte qu'un seul segment. La condition précédente rapportée au schéma suivant (figure 3.4) signifie dès lors

$$\sum_{i=1}^{10} A[i] = \prod_{i=1}^{10} A[i]$$

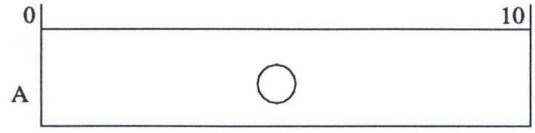


FIG. 3.4: Condition dans un tableau

Fonctions booléennes dans une expression

La condition "tricroiss or tridecroiss" dans le tableau de la figure 3.5 ci-dessous signifie que les éléments du segment sur lequel la condition porte sont triés de façon croissante ou décroissante.

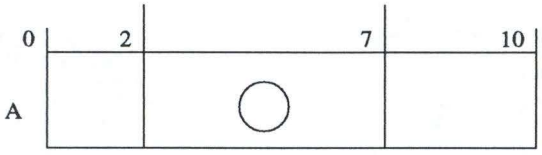


FIG. 3.5: Condition dans un segment

La condition "appartient X" dans le même tableau de la figure 3.5 signifie qu'un des éléments $A[3], \ldots, A[7]$ possède une valeur identique à celle de la variable X.

Considérons à présent le tableau suivant :

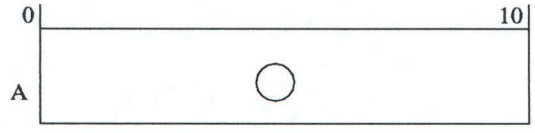


FIG. 3.6: Condition dans un tableau

La condition *egal* signifie que tous les éléments du tableau sont égaux :

$$A[1] = A[2] = \ldots = A[10]$$

La condition *inchange* signifie que le tableau possède les mêmes valeurs qu'avant l'exécution du programme. Si nous désignons par A_0 le tableau A à l'environnement initial, la condition peut se traduire par

$$\forall i \ : \ 1 \leq i \leq 10 : \ A_0[i] = A[i]$$

Dans le schéma de la figure 3.7,

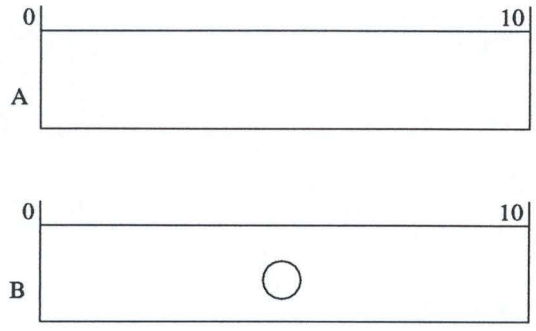


FIG. 3.7: Condition se rapportant à un autre tableau

la condition "*permute* A" signifie que les éléments du tableau B sont une permutation des éléments du tableau A.

Quantificateur universel dans une expression

Dans le tableau de la figure 3.8,

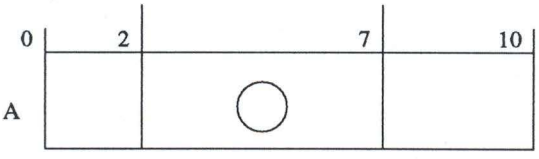


FIG. 3.8: Condition dans un segment

la condition "*pourtout* X : $X > 0$" signifie que tous les éléments du segment auquel la condition se rapporte sont strictement positifs.

Dans le même tableau de la figure 3.8, la condition "*pourtout* X : $X = Y$" signifie que tous les éléments du segment $A[3], \ldots, A[7]$ ont la même valeur que la variable Y :

$$\forall i \ : \ 3 \leq i \leq 7 \ : \ A[i] = Y$$

Quantificateur existentiel dans une expression

Dans le tableau de la figure 3.9,

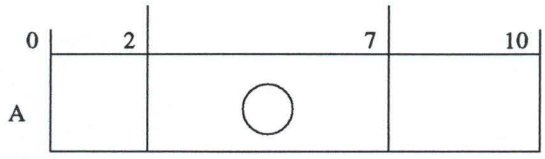


FIG. 3.9: Condition dans un segment

la condition "*ilexiste* X : $X > 0$" signifie qu'un élément du segment auquel la condition se rapporte est strictement positif.

Dans le même tableau de la figure 3.9, la condition "*ilexiste* X : $X <> Y$" signifie qu'il existe un élément du segment $A[3], \ldots, A[7]$ possédant une valeur différente de celle de la variable Y :

$$\exists i \ : \ 3 \leq i \leq 7 \ : \ A[i] <> Y$$

Chapitre 4

Guide d'utilisation

Dans ce chapitre sont décrites les fonctionnalités du programme d'aide à l'apprentissage de la programmation. Nous y présentons en détails les items du menu et les boîtes de dialogue permettant la saisie d'informations. Ensuite, nous montrons comment l'interface met en oeuvre la méthode des invariants.

Nous commençons par une description succincte du menu et ensuite décrivons dans la section 4.2 les fonctionnalités offertes par l'interface. Chaque fonctionnalité nécessitera le déclenchement d'une action du menu et bien souvent la saisie d'informations dans une boîte de dialogue. Dans la section 4.3, nous faisons le rapprochement avec la méthode des invariants.

4.1 Le menu principal

Au lancement du programme, la fenêtre de la figure 4.1 apparaît, on y aperçoit les items de la barre de menu décrits brièvement ici :

- Fichier : Le menu fichier permet de sortir de l'application.
- Situation : permet de passer d'une situation à une autre et d'accéder à la fenêtre de saisie du code. Il est également possible de changer la fenêtre active en cliquant sur l'onglet correspondant.
- Ajouter : permet l'ajout d'un tableau, d'une représentation, d'une borne, d'une variable et d'une condition.
- Supprimer : permet la suppression d'un tableau, d'une représentation, d'une borne, d'une variable et d'une condition.
- Initialiser : permet d'initialiser les tableaux et variables définis.

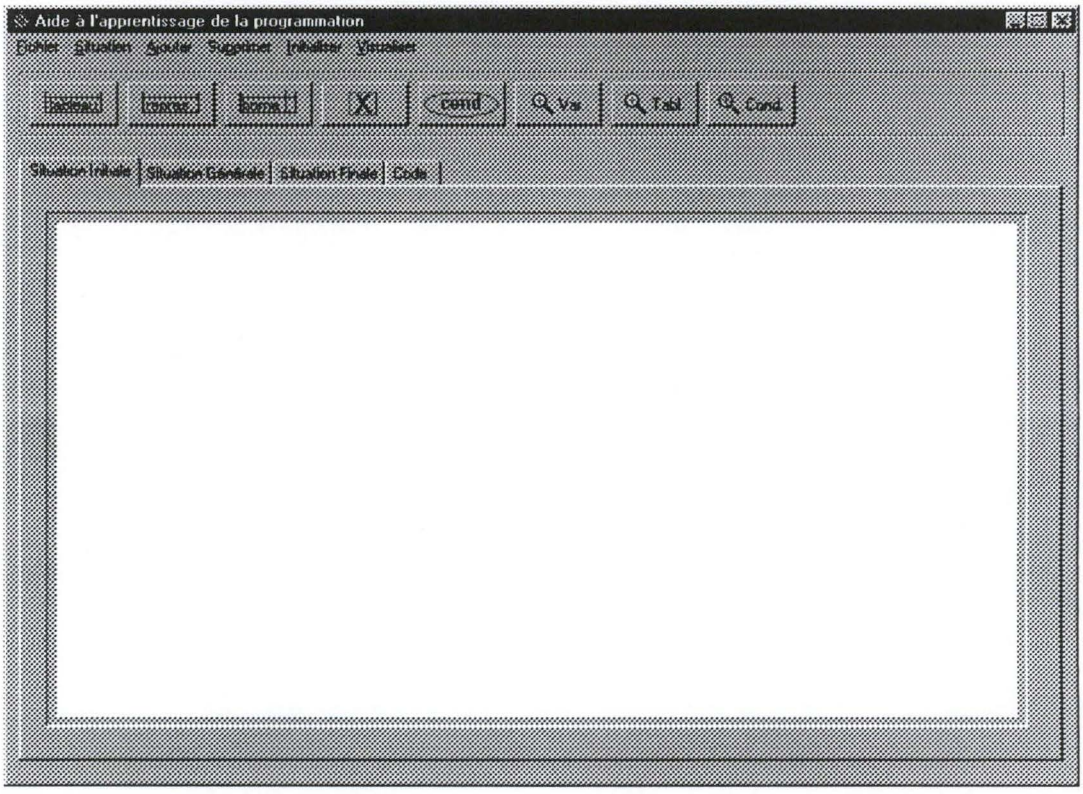


FIG. 4.1: Fenêtre principale

- Visualiser : permet de visualiser les valeurs des tableaux et variables définis. Il offre également la possibilité d'afficher les éventuelles conditions associées aux situations.

Sous la barre de menu figurent des boutons de raccourcis. Ils correspondent aux actions les plus souvent utilisées du menu. Il s'agit de :

- l'ajout d'un tableau,
- l'ajout d'une représentation,
- l'ajout d'une borne,
- l'ajout d'une variable,
- l'ajout d'une condition,
- la visualisation des valeurs des variables,
- la visualisation des valeurs des tableaux,

– la visualisation des conditions.

Une bulle d'information apparaît lorsque le curseur de la souris est immobilisé sur un de ces boutons afin de clarifier le type d'action qu'il déclenche.

Sous les boutons de raccourcis sont disposés quatre onglets. Les 3 premiers sont utilisés pour la représentation graphique des situations. C'est pourquoi ils ont l'aspect d'une feuille de dessin. Le dernier, montré à la figure 4.2, est utilisé pour entrer le code du programme utilisateur. Une boîte d'édition a été associée à chaque partie du programme utilisateur (Initialisation, Condition d'arrêt, Clôture et Itération). Une fois le code saisi, le bouton "Valider" effectue une vérification syntaxique. Rappelons que la syntaxe doit être conforme à celle du Pascal standard. Le bouton "Exécuter" lance le programme défini en vérifiant la validité des situations. Le bouton "Réinitialiser" offre à l'utilisateur la possibilité de reprendre les valeurs initiales des tableaux et variables définis dans le menu "Initialiser".

Remarquons que nous nous sommes limités à un sous-ensemble du langage Pascal, étant donné que les programmes d'apprentissage n'utilisent que quelques instructions élémentaires.

Les instructions permises sont les suivantes :

– l'instruction d'affectation,

– l'instruction "if then",

– l'instruction "if then else",

– l'instruction "while do".

Par contre, nous simulons entièrement les expressions.

4.2 Les fonctionnalités

Nous décrivons ici les fonctionnalités offertes par l'interface et la succession des actions que nécessite leur mise en oeuvre. Toutes les boîtes de dialogue apparaissant lors de la mise en oeuvre de ces fonctionnalités comportent des boutons "Annuler" et "Aide". Le premier annule l'opération en cours et le second fournit une aide sur la suite des actions à effectuer. Nous n'insisterons plus sur ces boutons par la suite.

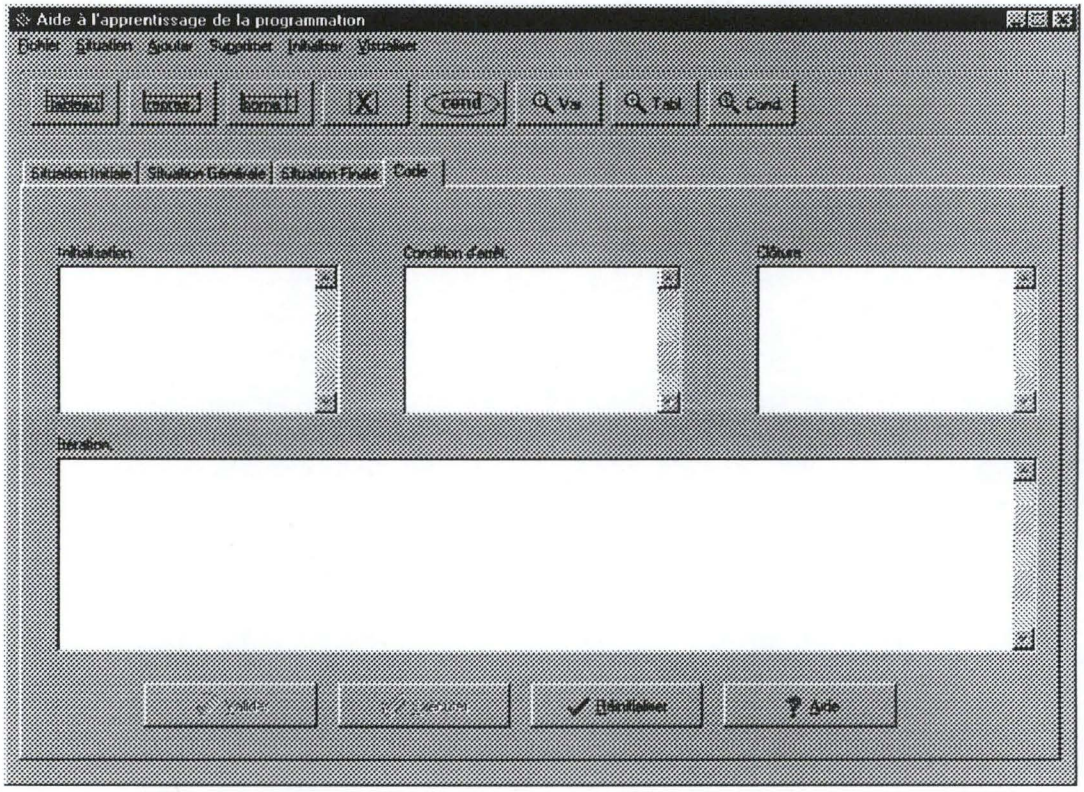


FIG. 4.2: Fenêtre principale

4.2.1 Fonctionnalité "Ajout d'un tableau"

Afin d'ajouter un tableau à une situation, l'utilisateur doit tout d'abord appeler le menu "Ajout" ; dans ce menu, il faut sélectionner l'item "Tableau". Une alternative consiste à utiliser le bouton de raccourci correspondant. Apparaît alors une boîte de dialogue montrée à la figure 4.3.

Dans cette boîte doivent être introduits le nom du tableau ainsi que les valeurs de ses bornes inférieure et supérieure.

La situation active est cochée par défaut. Si le tableau doit être ajouté dans une autre situation, on choisira la situation adéquate. En poussant sur le bouton "Ok", l'utilisateur peut dessiner son tableau sur la feuille blanche. Pour ce faire, il doit maintenir le bouton gauche de la souris enfoncé et tirer celle-ci vers la droite. Lorsque le bouton est relâché, le tableau se dessine

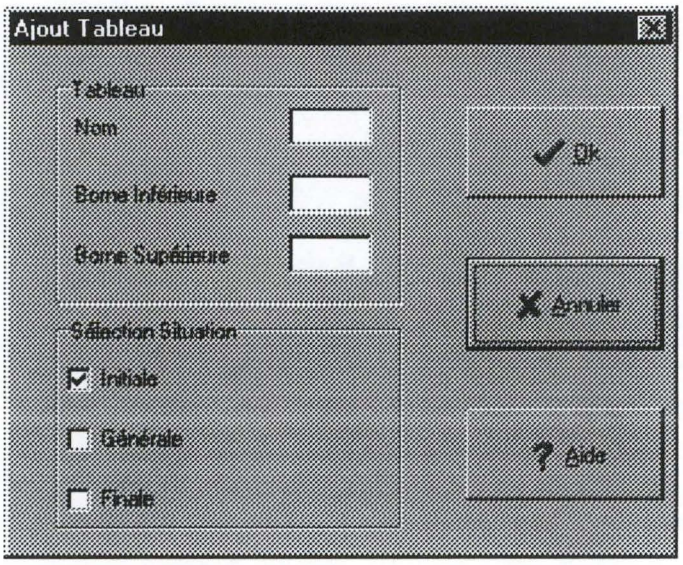


FIG. 4.3: Boîte de dialogue "Ajout Tableau"

entièrement.

Notons que pour une question de clarté du dessin, il est préférable d'utiliser des noms de tableau relativement courts.

4.2.2 Fonctionnalité "Ajout d'une représentation"

Pour ajouter une représentation, l'utilisateur doit appeler l'item "Représentation" du menu "Ajout". Une boîte de dialogue représentée à la figure 4.4 apparaît alors. Le bouton de raccourci associé produit bien sûr le même effet.

Dans la liste sont repris les tableaux de la situation active. Il suffit de sélectionner celui dont on désire une nouvelle représentation dans cette situation et de cliquer sur le bouton "Ok". Ensuite, il reste à dessiner selon le même procédé que celui explicité pour l'ajout d'un tableau.

L'application refuse l'ajout d'une représentation lorsqu'aucun tableau n'est défini dans la situation active.

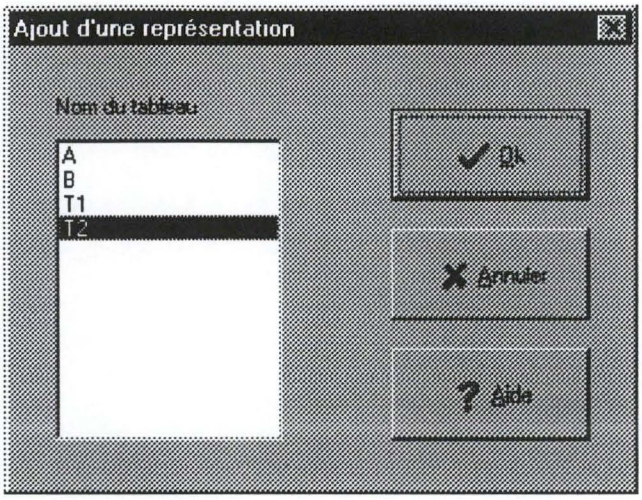


FIG. 4.4: Boîte de dialogue "Ajout d'une représentation"

4.2.3 Fonctionnalité "Ajout d'une borne"

L'ajout d'une borne s'effectue en choisissant l'item "Borne" du menu "Ajout", ce qui fait apparaître la boîte de dialogue de la figure 4.5. L'utilisateur dispose ici aussi d'un bouton de raccourci.

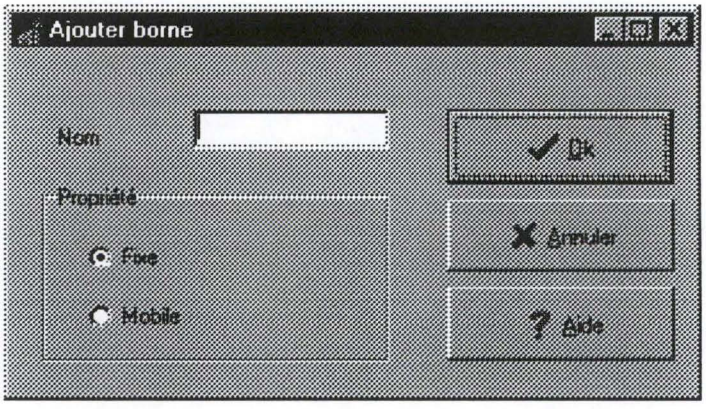


FIG. 4.5: Boîte de dialogue "Ajout d'une borne"

L'utilisateur doit choisir le caractère fixe ou mobile de la borne. Une borne

fixe est obligatoirement associée à une variable [1], tandis qu'une borne mobile peut bouger à l'intérieur du bloc ou du tableau dans lequel elle se trouve (cfr section 3.3.10).

Un segment peut être défini comme une partie de tableau délimitée par deux bornes.

Après avoir appuyé sur le bouton "Ok", il reste à placer la borne à l'endroit désiré. Pour cela, il faut cliquer à l'intérieur d'une représentation de tableau. Dans le cas contraire, la borne ne sera pas placée et l'opération sera annulée.

Ici aussi, nous vérifions qu'au moins un tableau est défini dans la situation active avant le déclenchement de la fonctionnalité.

4.2.4 Fonctionnalité "Ajout d'une variable"

Pour ajouter une variable, il faut sélectionner l'item "Variable" du menu "Ajout". La boîte de dialogue reprise à la figure 4.6 s'ouvre alors. Rappelons qu'un bouton de raccourci est associé à cet item.

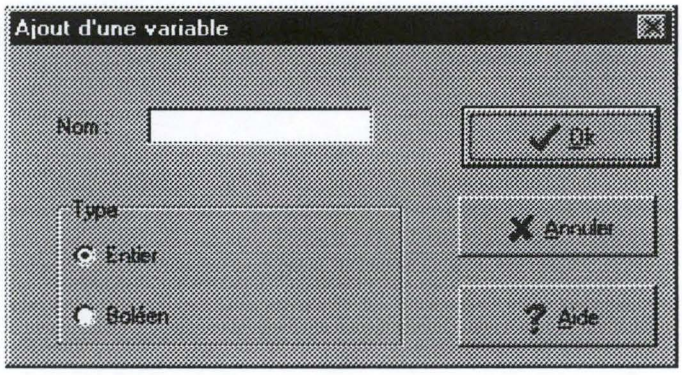


FIG. 4.6: Boîte de dialogue "Ajout d'une variable"

Cette boîte permet la saisie du nom et du type de la variable (entier ou booléen). Une fois ces renseignements complétés, le bouton "Ok" a pour effet de déclarer la variable au programme.

[1] Si la variable associée à la borne fixe n'a pas encore été déclarée, le système la déclare.

4.2.5 Fonctionnalité "Ajout d'une condition"

L'ajout d'une condition est mis en oeuvre par l'item "Condition" du menu "Ajout" ou le bouton de raccourci associé. Cette action ouvre la boîte de dialogue représentée à la figure 4.7 ci-dessous.

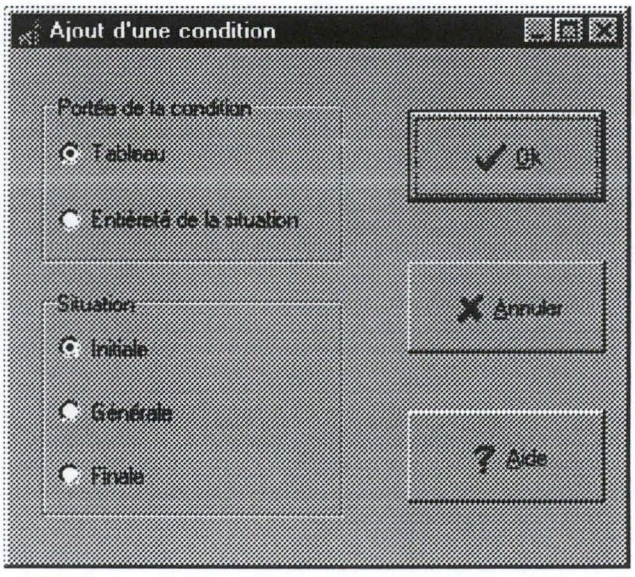


FIG. 4.7: Boîte de dialogue "Ajout d'une condition"

Elle permet de déterminer si la condition se rapporte à l'entièreté d'une situation ou à un tableau d'une situation. On y détermine également la situation où la condition doit être ajoutée. En appuyant sur "Ok", la boîte de dialogue montrée à la figure 4.8 apparaît.

C'est à travers elle qu'est introduite la condition. Cette dernière doit vérifier la syntaxe concrète du langage graphique défini au chapitre 3. La syntaxe peut être rappelée en cliquant sur le bouton "Aide" de cette même fenêtre.

Le bouton "Ok" lance un parseur qui permet de détecter les éventuelles erreurs de syntaxe. Si une erreur a effectivement été trouvée, celle-ci est signalée à l'utilisateur. Dans le cas contraire, soit la condition est ajoutée directement si l'option "Entièreté de la situation" avait été choisie dans la

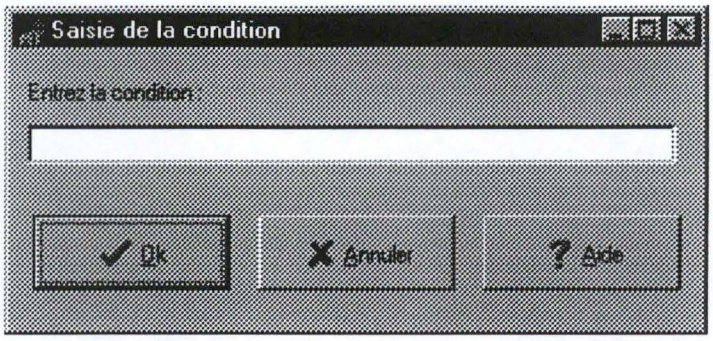


FIG. 4.8: Boîte de dialogue "Saisie de la condition"

fenêtre 4.7, soit l'utilisateur doit encore la placer. Pour cela, il lui suffit de cliquer avec le bouton gauche de la souris dans un segment de tableau d'une des trois situations.

4.2.6 Fonctionnalité "Suppression d'un tableau"

La suppression d'un tableau d'une des situations s'effectue de la façon suivante.

Dans un premier temps, il faut activer la situation pour laquelle l'opération doit être effectuée. Ensuite, l'item "Tableau" du menu "Supprimer" doit être sélectionné. Si au moins un tableau figure dans la situation active, la boîte de dialogue reprise à la figure 4.9 est affichée. Dans le cas contraire, l'opération de suppression est annulée.

Il reste à choisir le nom du tableau que l'on désire supprimer. A cet effet, on utilise la liste de sélection sous le libellé "Nom du tableau". La validation grâce au bouton "Ok" supprime de la situation active toutes les représentations du tableau sélectionné.

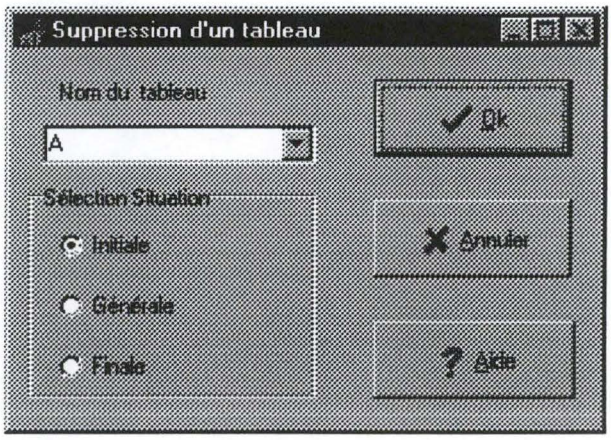


FIG. 4.9: Boîte de dialogue "Suppression d'un tableau"

4.2.7 Fonctionnalité "Suppression d'une représentation"

Pour supprimer une représentation, on utilise l'item "Représentation" du menu "Supprimer". Si la situation active comporte au moins un tableau, la boîte de dialogue illustrée à la figure 4.10 est affichée. Dans le cas contraire, l'opération de suppression est annulée.

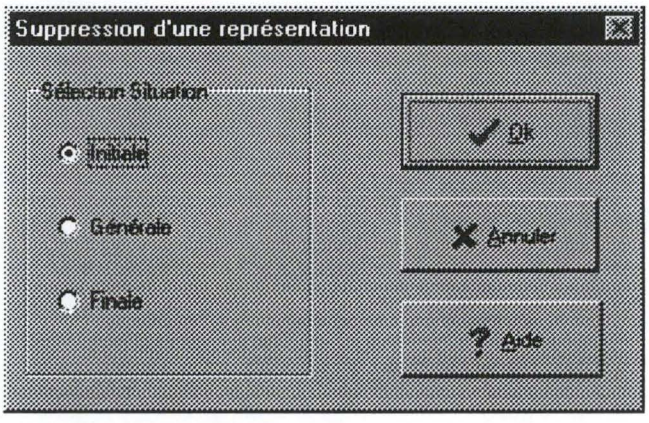


FIG. 4.10: Boîte de dialogue "Suppression d'une représentation"

L'utilisateur a la possibilité de changer de situation à l'intérieur de cette boîte grâce aux boutons radio. Lorsque le choix de la situation et la validation (bouton "Ok") ont eu lieu, la suppression de la représentation s'opère après

avoir cliqué obligatoirement dans celle-ci.

4.2.8 Fonctionnalité "Suppression d'une borne"

La suppression d'une borne s'effectue en activant le menu "Supprimer" et en sélectionnant dans celui-ci l'item "Borne". Dans la boîte de dialogue représentée à la figure 4.11, il faut sélectionner la situation dans laquelle l'opération doit être effectuée. Après avoir cliqué sur le bouton "Ok", il ne reste plus qu'à cliquer du bouton gauche de la souris sur la borne à supprimer. Celle-ci s'efface ensuite du dessin. Si aucune borne n'est associée à la situation, l'opération de suppression est annulée.

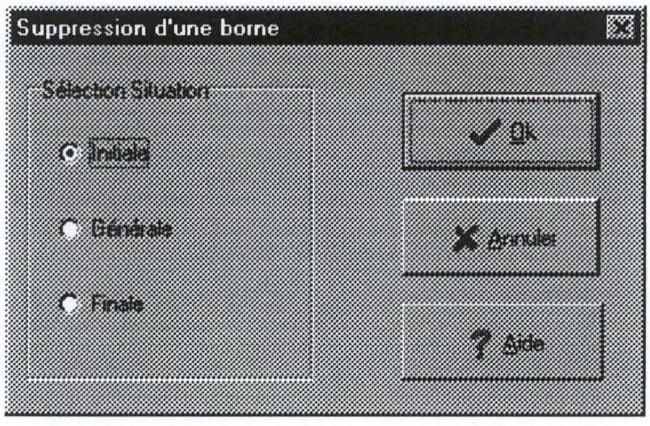


FIG. 4.11: Boîte de dialogue "Suppression d'une borne"

4.2.9 Fonctionnalité "Suppression d'une variable"

Pour supprimer une variable, il suffit d'activer le menu "Supprimer" et de sélectionner l'item "Variable". Dans la boîte de dialogue qui apparaît ensuite et qui est représentée à la figure 4.12, il reste à sélectionner le nom de la variable à enlever. Le bouton "Ok" permet de valider l'opération.

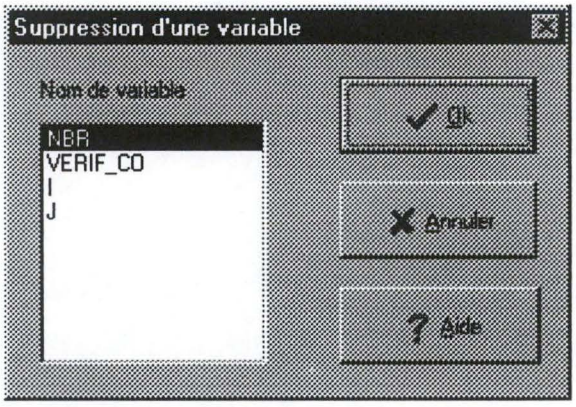


FIG. 4.12: Boîte de dialogue "Suppression d'une variable"

4.2.10 Fonctionnalité "Suppression d'une condition"

Afin de supprimer une condition, l'item "Condition" du menu "Supprimer" doit être sélectionné. Dans la boîte de dialogue reprise à la figure 4.13, l'utilisateur sélectionne la situation devant être modifiée. Ensuite il suffit de cliquer du bouton gauche de la souris à l'intérieur du cercle schématisant la condition à retirer.

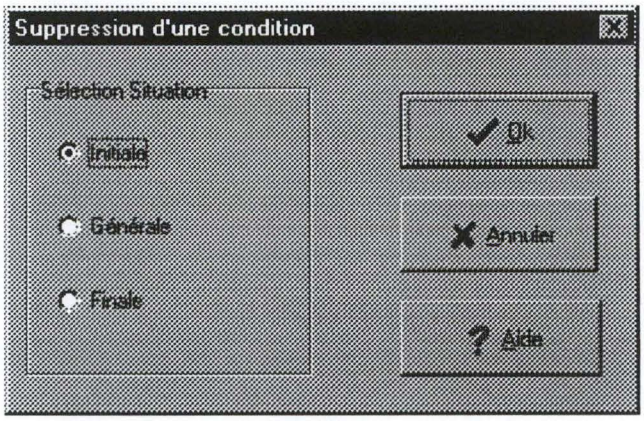


FIG. 4.13: Boîte de dialogue "Suppression d'une condition"

4.2.11 Fonctionnalité "Initialisation des variables"

L'initialisation des variables s'effectue en choisissant l'item "Variable" du menu "Initialiser". Le fenêtre illustrée à la figure 4.14 reprend toutes les variables qui ont été définies ainsi que leur type. En cliquant sur leur nom, leur valeur apparaît sous le libellé "Variable sélectionnée". Il est possible de modifier la valeur de chaque variable en entrant une nouvelle valeur dans le champ d'édition.

Signalons que pour chaque variable, la nouvelle valeur n'est enregistrée que lorsque le bouton "Modifier" est pressé. De plus, avant enregistrement de la nouvelle valeur, le programme vérifie que la valeur introduite respecte bien le type de la variable sélectionnée. Le bouton "Fermer" permet de retourner à la fenêtre principale.

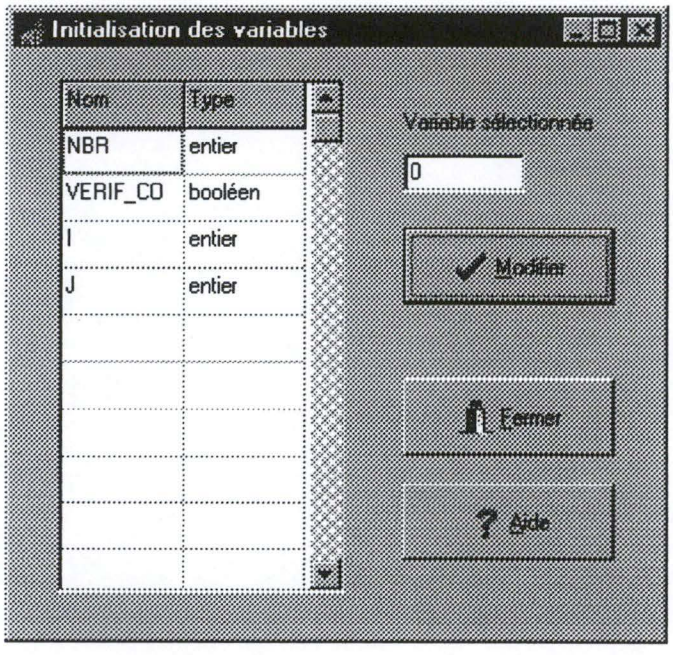


FIG. 4.14: Boîte de dialogue "Initialisation des variables"

4.2.12 Fonctionnalité "Initialisation des tableaux"

Pour initialiser un tableau, on utilise l'item "Tableau" du menu "Initialiser". Cette action déclenche l'ouverture de la fenêtre représentée à la figure 4.15. Dans celle-ci, on aperçoit une liste de sélection déroulante permettant de choisir le tableau à initialiser. Les bornes inférieure et supérieure du tableau sélectionné sont automatiquement mises à jour. Afin de modifier les valeurs des éléments du tableau choisi, on se sert des champs d'édition "indice" et "valeur".

La modification des valeurs d'un tableau ne prend cours que lorsque le bouton "Modifier" est enclenché. Si un autre tableau est sélectionné avant cette action, les données seront perdues. Par contre, la vérification du type est effectuée chaque fois que l'indice est modifié. Si une erreur se produit, l'ancienne valeur est conservée. Le bouton "Fermer" permet de retourner à la fenêtre principale.

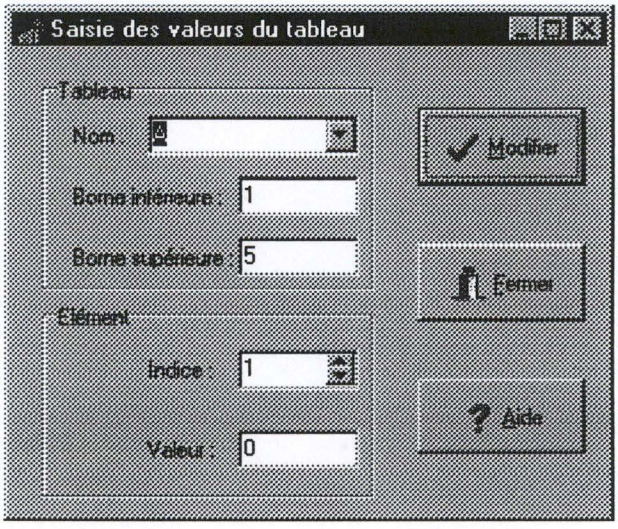


FIG. 4.15: Boîte de dialogue "Initialisation des tableaux"

4.2.13 Fonctionnalité "Visualisation des variables"

Cette fonctionnalité est réalisée par sélection de l'item "Variable" du menu "Visualiser". Cette action ouvre la fenêtre illustrée à la figure 4.16.

Une autre possibilité est d'utiliser le bouton de raccourci prévu à cet effet.

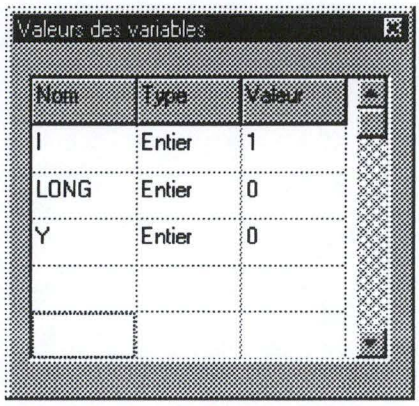


FIG. 4.16: Boîte de dialogue "Valeurs des variables"

4.2.14 Fonctionnalité "Visualisation des tableaux"

La visualisation des valeurs des éléments d'un tableau est possible en sélectionnant l'item "Tableau" du menu "Visualiser" ou en cliquant sur le bouton de raccourci prévu à cet effet. Il en résulte l'ouverture de la fenêtre montrée à la figure 4.17.

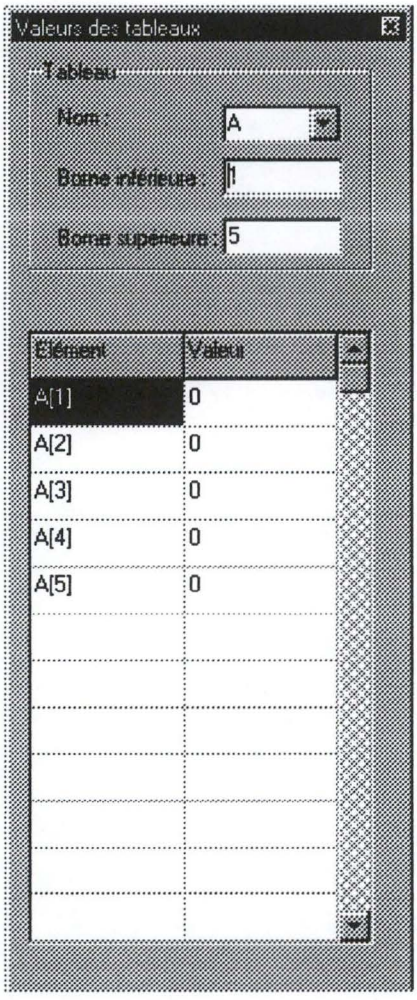


FIG. 4.17: Boîte de dialogue "Valeurs des tableaux"

4.2.15 Fonctionnalité "Visualisation des conditions"

Une condition peut être affichée par sélection de l'item "Condition" du menu "Visualiser" ou grâce au bouton de raccourci associé. Un message indique à l'utilisateur qu'il doit alors cliquer à l'intérieur d'un cercle représentant une condition. Un exemple de fenêtre donnant l'expression d'une condition est donné à la figure 4.18.

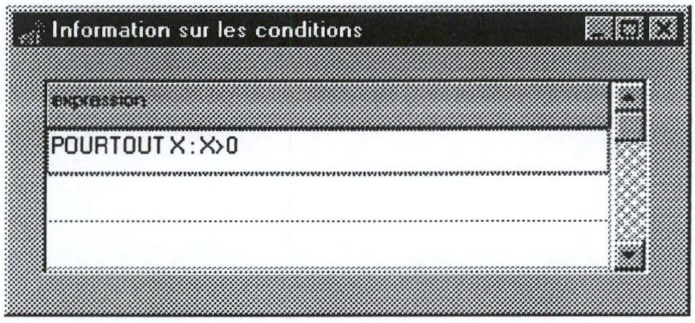


FIG. 4.18: Boîte de dialogue "Information sur les conditions"

4.3 Lien avec la méthode des invariants

Nous explicitons ici comment les concepts de la méthode présentée au chapitre 1 ont été transposés dans l'interface de l'outil d'aide à la programmation.

Le concept de schéma

Le concept de schéma est l'équivalent de la notion de représentation de l'interface. Afin d'ajouter un schéma à une situation, l'utilisation de la fonctionnalité "Ajout d'une représentation" est nécessitée. Ceci est vrai dans le cas où le schéma porte le nom d'un tableau qui a déjà été dessiné. Dans le cas contraire, il faudra utiliser la fonctionnalité "Ajout d'un tableau". En effet, lors de l'ajout d'un tableau, c'est en réalité une représentation de ce tableau qui est placée.

Le concept de segment

Le concept de segment est absent de l'interface. Celui-ci n'était pas nécessaire puisqu'un segment est défini par deux bornes et une représentation. La notion de borne est donc bien présente. Afin de définir les segments, on utilisera la fonctionnalité "Ajout d'une borne".

Les conditions portant sur les segments sont introduites grâce à la fonctionnalité "Ajout d'une condition".

Situations et code du programme

Les différentes situations (initiale, générale et finale) sont accessibles à partir d'onglets et ont l'apparence d'une planche à dessin. Le code est aussi associé à un onglet où l'on distingue quatre boîtes d'édition qui correspondent aux parties du code identifiées dans la méthode (Init, Iter, Clôt, B).

Chapitre 5

Exemples d'utilisation

Dans ce chapitre, nous allons donner des exemples d'utilisation du système. Le but du premier exemple est de montrer que l'on peut spécifier un problème ne comportant pas de tableau. Le deuxième fait intervenir des tableaux et des bornes fixes. Le dernier comporte toutes les notions du langage L.D.S et nous montre une réaction du système face à des erreurs.

5.1 Exemple 1 : Calcul du carré d'un entier

5.1.1 Enoncé du problème

Il est demandé à l'étudiant de calculer le carré d'un entier. Celui-ci sera fourni dans la variable `entree` et le résulat placé dans la variable `sortie` en utilisant la relation $(x+1)^2 = x^2 + 2x + 1$.

5.1.2 Spécifications

Liste des objets utilisés

Dans ce problème, l'étudiant n'utilisera que des variables entières dont voici la liste :

Variable	type	Commentaires
`entree`	entier	contient l'entier dont on veut calculer le carré
`sortie`	entier	contient le résultat
`i`	entier	indice
`y`	entier	variable intermédiaire
`z`	entier	variable intermédiaire

Les variables sont introduites par l'intermédiaire de la fenêtre de la figure 5.1. Cette fenêtre est activée par le menu "Ajouter-Variable".

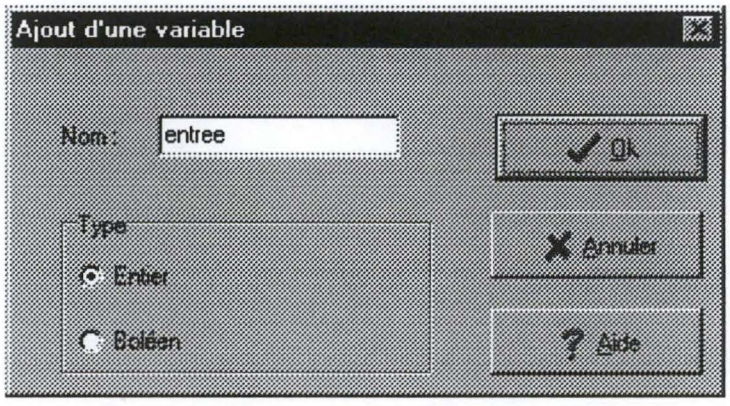


FIG. 5.1: Ajout d'une variable

Situation Initiale

La condition porte sur la variable `entree` qui doit être positive. Voici comment l'utilisateur introduit cette condition.

- étape 1 : il presse le bouton "Ajout d'une condition",
- étape 2 : il sélectionne la portée de la condition (entièreté de la situation), voir figure 5.2,
- étape 3 : il encode la condition dans la fenêtre de la figure 5.3.

Situation Finale

Les conditions portent sur les variables `entree` et `sortie`. Nous explicitons les assertions :

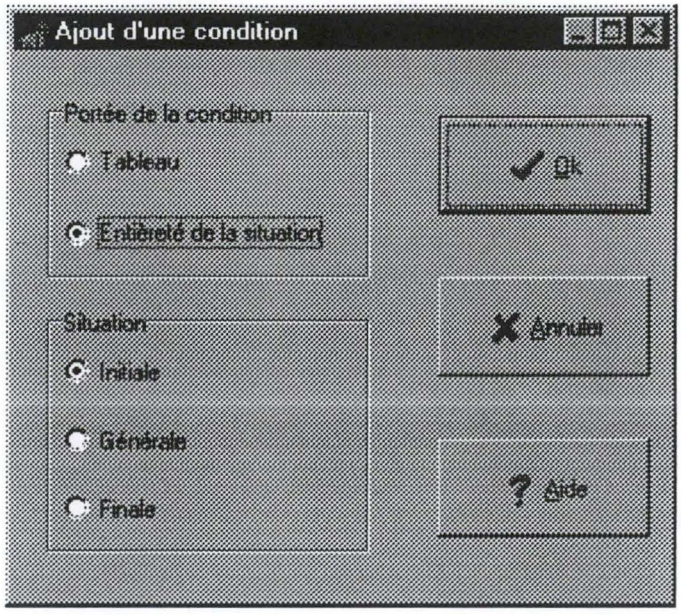


FIG. 5.2: Sélection de la portée de la condition

- `entree` est inchangé,
- `sortie` vaut `entree` au carré.

5.1.3 Situation générale et condition d'arrêt

La situation générale est décrite par les trois conditions suivantes :

- $0 \leq \texttt{i} \leq \texttt{entree}$
- $\texttt{y} = \texttt{i}^2$
- z=2*i+1

La condition d'arrêt est que `i` soit égale à `entree`.

$$B : \text{i=entree}$$

5.1.4 Les différentes suites d'instructions

Voici les différentes suites d'instructions :

Init
i :=0;

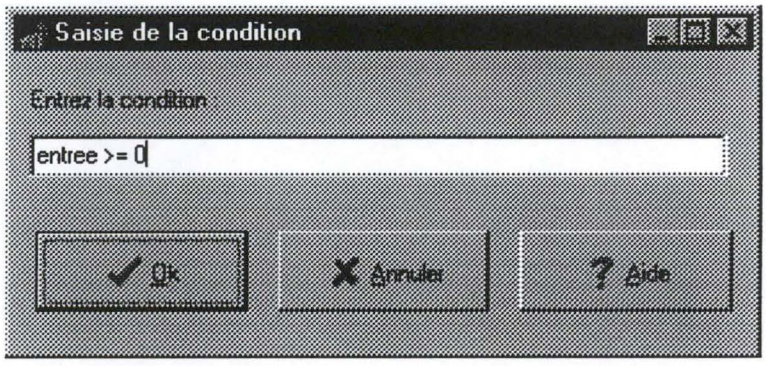


FIG. 5.3: Introduction de la condition

z :=1 ;
y :=0
Iter
i :=i+1 ;
y :=y+z ;
z :=z+2
Clot
sortie :=y

L'utilisateur encode les différentes suites d'instructions dans la fenêtre de la figure 5.4

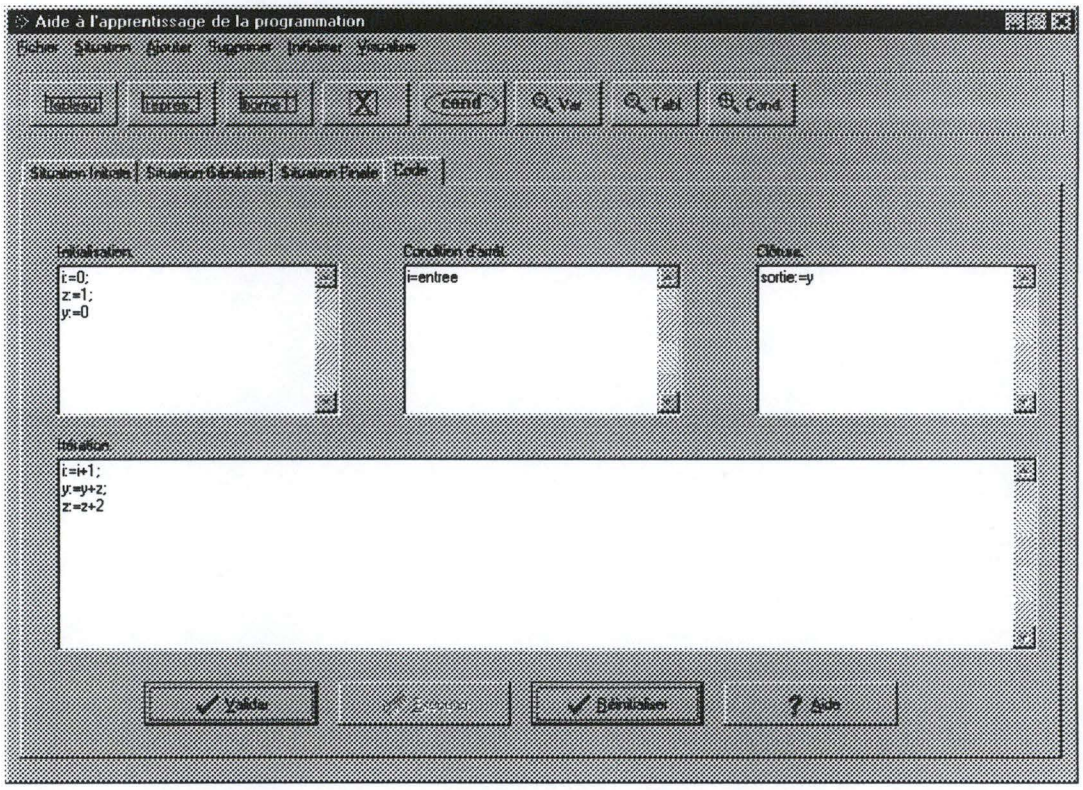


FIG. 5.4: Introduction des instructions et de la condition d'arrêt

5.1.5 Résultats

Plutôt qu'un long discours nous présentons les résultats sous forme d'une fenêtre que voici :

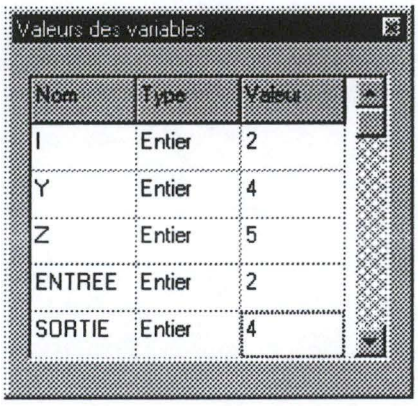


FIG. 5.5: Résultats de l'exemple 1

On voit bien à la figure 5.5 que la variable `sortie` est le carré de `entree`.

5.2 Exemple 2 : Calcul de la somme des éléments d'un tableau

5.2.1 Enoncé du problème

Il est demandé à l'étudiant de calculer la somme des éléments d'un tableau. Le résultat sera stocké dans une variable entière nommée S.

5.2.2 Spécifications

Liste des objets utilisés

Nous commençons par les tableaux (ceux-ci seront déclarés implicitement lors de la création des graphiques)

Tableau	Commentaires
A	tableau initial

Pour inclure un tableau, l'utilisateur appelle la fenêtre de la figure 5.6 par le menu "Ajouter-Tableau". Ensuite, il dessine le tableau comme expliqué à la section 4.2.1.

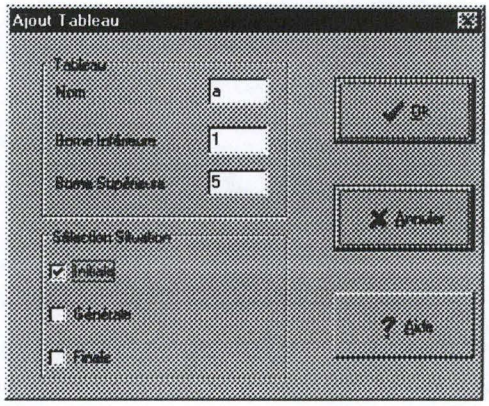


FIG. 5.6: Ajout d'un tableau

Les variables devront être déclarées explicitement (sauf si elles sont associées à une borne fixe).

Variable	type	Commentaires
S	entier	contient le résultat de la somme
i	entier	indice de boucle

Situation Initiale

La seule condition porte sur le tableau `A` qui doit être initialisé. Ce que nous représentons par :

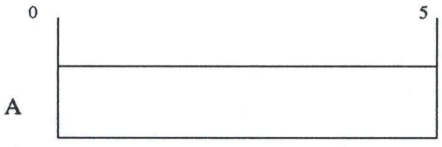


FIG. 5.7: Situation Initiale du problème 2

Situation Finale

La situation finale affirme que le tableau `A` est inchangé et que la variable `S` vérifie l'égalité suivante : `S`$= \sum_{i=1}^{5} A[i]$.

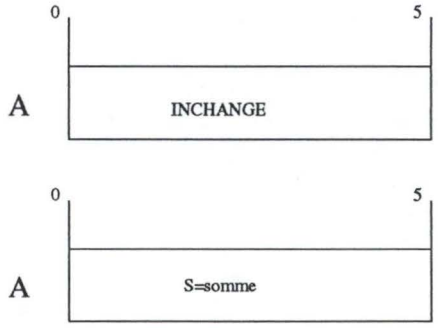


FIG. 5.8: Situation Finale du problème 2

A partir de maintenant, les conditions seront portées dans le dessin pour plus de clarté.

5.2.3 Situation générale et condition d'arrêt

Dans la situation générale, nous devons exprimer le fait que le tableau `A` est inchangé et que la somme de la partie déjà parcourue se trouve dans `S` (voir figure 5.9).

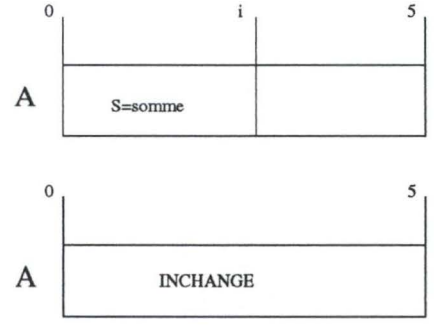


FIG. 5.9: Situation Générale du problème 2

La condition d'arrêt est que la variable `i` soit arrivée en fin de course c'est-à-dire qu'elle vale 5.

B : i=5

5.2.4 Les différentes suites d'instructions

Les différentes suites d'instructions sont données par :

Init
i :=0 ;
S :=0

Iter
i :=i+1 ;
S :=a[i]+S

Clot
* aucune* [1]

5.2.5 Résultats

Le système indique que la syntaxe est correcte et que tous les objets sont déclarés. Il confirme que les relations vues au paragraphe 1.2.3 sont vérifiées et indique que le programme est correct. De plus, si l'utilisateur introduit un tableau A de la forme suivante :

Tableau A	
Indice	Valeur
1	5
2	4
3	3
4	2
5	1

Le système lui fournira comme résultat S=15. ce que nous vérifions dans la fenêtre suivante.

[1] A partir de maintenant, nous mettrons les commentaires entre astérisques, bien que ceux-ci ne fassent pas partie de la syntaxe.

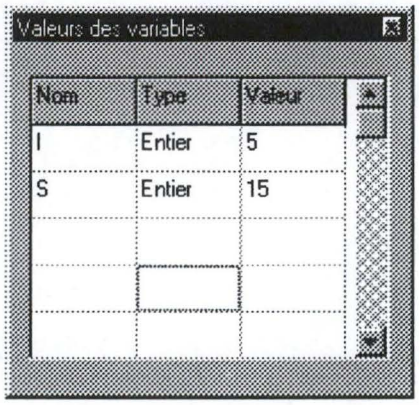


FIG. 5.10: Tableau A

5.3 Exemple 3 : Calcul du plus long plateau

5.3.1 Enoncé du problème

Il est demandé à l'étudiant de calculer la longueur du plus long plateau composé d'entiers identiques dans un tableau.

5.3.2 Spécifications

Liste des objets utilisés

Nous commençons comme d'habitude par les tableaux et nous enchaînons avec les variables.

Tableau	Commentaires
`A`	tableau initial

Et maintenant les variables,

Variable	type	Commentaires
`long`	entier	contient la longueur du plus long plateau courant
`y`	entier	contient la longueur du plateau courant
`i`	entier	indice

Situation Initiale

La seule condition porte sur le tableau `A` qui doit être initialisé. Ce que nous représentons par :

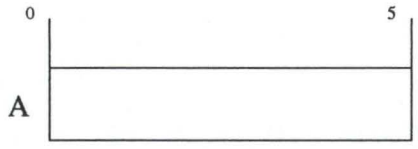


FIG. 5.11: Situation Initiale du problème 3

Situation Finale

Cette situation est composée des conditions suivantes :

1. le tableau `A` est inchangé
2. `long` est la longueur du plus long plateau composé d'entiers identiques.

Nous paraphrasons la deuxième condition en disant que tout autre plateau composé d'entiers identiques a une longueur inférieure ou égale à `long`. Graphiquement, nous obtenons

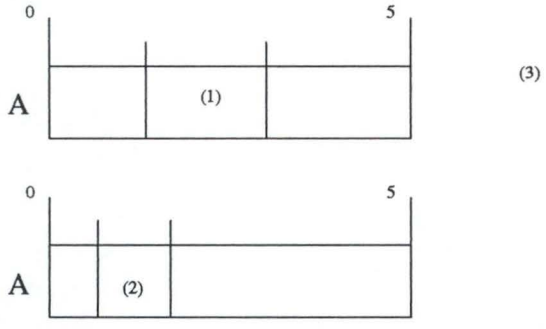


FIG. 5.12: Situation Finale du problème 3

Dans la figure 5.12, les conditions sont :

Conditions		
(1)	(2)	(3) (associée à la situation)
egal longueur=`long`	egal longueur=y	$long \geq y$

Nous voyons que pour exprimer les conditions portant sur plusieurs segments distincts, nous avons dû introduire deux variables intermédiaires, à savoir `long` et `y`. Une formulation plus intuitive aurait été `longueur` $\geq$ `longueur(Y)`. Malheureusement, notre langage ne la permet pas.

5.3.3 Situation Générale et condition d'arrêt

La situation générale nous indique que la longueur du plateau courant est en permanence inférieure à `long`. Ce que nous exprimons graphiquement par la figure 5.13

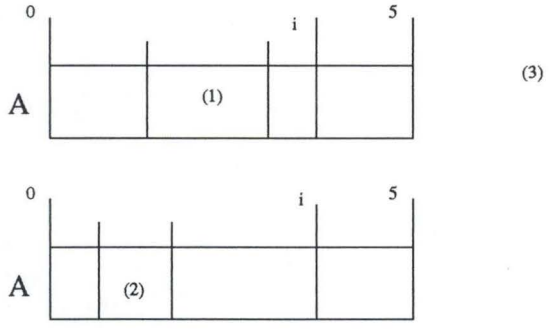


FIG. 5.13: Situation Générale du problème 3

Les conditions associées sont identiques à celles associées à l'illustration 5.12. La condition d'arrêt est que l'indice `i` vale 5.

B : i=5

5.3.4 Les différentes suites d'instructions

Les différentes suites d'instructions sont données par :

Init
i :=1;

long :=1 ;
y :=1

Iter
if a[i]=a[i+1] then y :=y+1 else y :=1 ;
if long < y then long :=y ;
i :=i+1

Clot
* aucunes*

5.3.5 Résultats

Pour avoir un résultat valable, il faut introduire un tableau `A` quelconque. Supposons que l'utilisateur ait introduit le tableau suivant :

Indice	Valeur
1	5
2	4
3	4
4	2
5	1

Comme le code est correct, et que l'enchaînement "code-condition" vérifie les relations 1.2.3, le système affecte la valeur 2 à la variable `long`. De plus, il affiche les résultats dans la fenêtre de la figure 5.14 :

5.3.6 Exemple fautif

Nous allons expliciter, dans ce paragraphe, les réactions du système lorsque le code ne correspond pas aux situations décrites.

Supposons que l'étudiant ait gardé les mêmes descriptions de situation, et qu'il ait introduit le code suivant :

Init
i :=0 ;
long :=0 ;
y :=0

Iter
i :=i+1 ;
if a[i]=a[i+1] then y :=y+1 else y :=0 ;

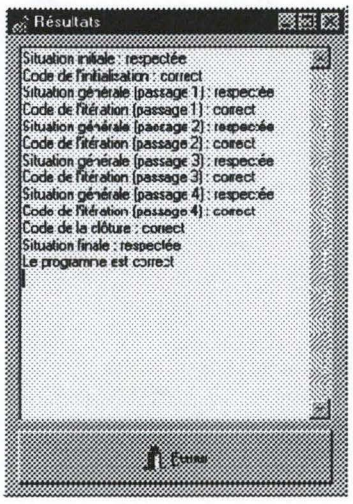


FIG. 5.14: Résultats fournis par le système

if long < y then long :=y

Clot
* aucunes*

B
i=4

Alors s'il encode le même tableau que celui du point 5.3.5, il obtient le message suivant :

FIG. 5.15: Mauvaise implémentation

Il est évident que la situation initiale est vérifiée (`A` est initialisé). Ensuite, le système vérifie la relation

$$\{S.I.\} \text{ Init } \{S.G.\}$$

qui est correcte (Il l'indique en affichant "Situation Générale (passage 1) : respectée"). Il va maintenant appliquer la relation

$$\{S.G. \wedge \neg B\} \text{ Iter } \{S.G.\}$$

Comme la situation générale est vérifiée mais pas la condition d'arrêt , il va appliquer le code d'itération et vérifier si la S.G. est remplie. Cette dernière est incorrecte car lorsqu'on analyse la figure 5.16, on constate que la variable `long` et l'indice `i` valent respectivement 0 et 1. Ces deux faits contredisent la situation décrite au paragraphe 5.3.3 qui affirme que `long` est la longueur du plus long segment constitué d'éléments identiques, cette variable doit donc être égale à 1. Remarquons que la longueur du plateau courant représentée

par y vaut 0 alors qu'elle devrait valoir 1. Mais comme le système s'arrête dès qu'il constate une erreur, il ne signale pas cette dernière.

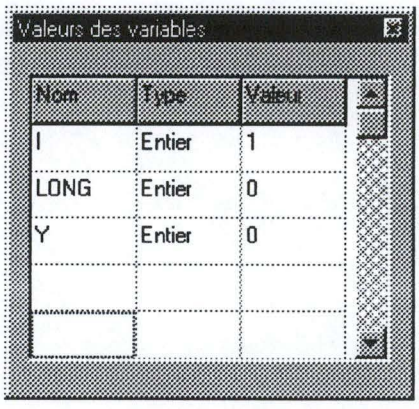


FIG. 5.16: Visualisation des variables

Chapitre 6

Implémentation

Dans ce chapitre sont présentées les idées générales employées lors de la conception de l'outil d'aide à l'apprentissage de la programmation. Nous nous limitons aux parties clés du programme, une présentation détaillée étant impossible à réaliser vu le nombre de lignes du listing (±7500) et d'autre part, ne serait guère intéressante.

Nous focaliserons notre attention sur les trois parties centrales du projet à savoir :

- la construction de l'arbre syntaxique des situations,
- la construction de l'arbre syntaxique des expressions et instructions,
- la sémantique des situations et instructions.

Celles-ci sont expliquées ci dessous. Le langage utilisé fut Delphi 2.0. Nous décrivons ses avantages et inconvénients à la fin de ce chapitre.

6.1 Arbre syntaxique des situations

6.1.1 Structure de données

Dans un premier temps, voyons quelle structure de données nous avons défini pour mémoriser la description des situations et évaluer celles-ci, i.e. tester leur validité :

Type description

```
description = class(TObject)
              sit_init : Psit;
              sit_gen : Psit;
              sit_fin : Psit;
              list_var : Plist_var;
              list_var_ini : Plist_var;
              list_tabl_val_ini : Plist_tabl2;
              list_tabl_val : Plist_tabl2;
              constructor create(_sit_init,_sit_gen,_sit_fin :
              Psit; _list_var,_list_var_ini : Plist_var;
              _list_tabl_val_ini, _list_tabl_val : Plist_tabl2);
              end;
```

L'objet Delphi `description` réalise cet objectif. Il contient les trois situations qui sont des pointeurs de type `Psit` vers un objet de type `sit`, deux pointeurs vers une liste de variables (de type `Plist_var`) et deux pointeurs vers une liste de tableaux (de type `Plist_tabl2`). Le constructeur `create` permet d'initialiser l'instance de l'objet `description`.

Le champ `list_var` est un pointeur vers la liste des variables. Cette liste contient pour chaque variable, son nom, son type et sa valeur. La liste `list_var_ini` conserve les valeurs définies lors de l'initialisation. Cette copie permettra par la suite à l'utilisateur de les réinitialiser après l'exécution de son programme.

Le champ `list_tabl_val` est un pointeur vers la liste des tableaux. Elle contient pour chacun d'entre eux, son nom, sa borne inférieure et sa borne supérieure, ainsi que l'ensemble de ses valeurs. Cette liste manipule des éléments de type `tabl2`. Ci-dessous se trouve un descriptif des types `tabl2` et `sit` qui servent à stocker respectivement un tableau et une situation.

Type tabl2

```
tabl2 = class(Tobject)
        nomtab : string;
        bi,bs : integer;
        list_valeur : Plist_valeur;
```

```
        constructor create(_nomtab : string; _bi,_bs : integer;
        _list_valeur : Plist_valeur);
        end;
```

Le champ `list_valeur` est un pointeur vers la liste des valeurs du tableau. Nous ne la détaillerons pas davantage, un développement plus profond n'étant pas nécessaire à la compréhension. Voyons plutôt les données englobées dans le type `sit`.

Type sit

```
sit = class(TObject)
      list_tabl : Plist_tabl;
      list_cond : Plist_cond;
      constructor create(_list_tabl : Plist_tabl; _list_cond :
      Plist_cond);
      end;
```

Cet objet doit permettre de mémoriser l'entièreté d'un dessin relatif à une situation (champ `list_tabl`) ainsi que les conditions relatives à celle-ci (champ `list_cond`). Le type `Plist_tabl` est un pointeur vers une liste d'objets du type `tabl`, ce dernier étant décrit plus loin. Le type `Plist_cond` est un pointeur vers une liste d'objets du type `expr_bool` :

```
expr_bool = class(Tobject)
            expr : Pconstr;
            constructor create(_expr : Pconstr);
            end;
```

Le champ `expr` est un pointeur vers la condition (de type `constr`). La façon de mémoriser les conditions sera expliquée plus loin, dans la section 6.2 relative à l'arbre syntaxique des expressions et instructions.

Type tabl

Chaque dessin de tableau comporte un nom, une borne inférieure et une borne supérieure. Nous mémorisons également à ce niveau toutes les représentations d'un tableau. Les représentations sont placées dans une liste accessible à partir du champ `list_repres` et contenant des objets de type `repres`.

```
tabl = class(Tobject)
       nomtab : string;
       bi,bs : integer;
       list_repres : Plist_repres;
       constructor create(_nomtab : string; _bi,_bs : integer;
       _list_repres : Plist_repres);
       end;
```

Type repres

Chaque représentation est identifiée par un nom, généré par le système. Les quatre champs suivants reprennent les coordonnées du rectangle sur la planche à dessin. Le dernier champ, `list_seg`, pointe vers la liste des bornes définies à l'intérieur de cette représentation. Chaque élément de cette liste est du type `seg`.

```
repres = class(Tobject)
         nom : string;
         hautgauchex,hautgauchey,basdroitx,basdroity :integer;
         list_seg : Plist_seg;
         constructor create(_nom : string; _list_seg : Plist_seg;
         _hautgauchex,_hautgauchey,_basdroitx,_basdroity :integer);
         end;
```

Type seg

Un nom identifiant est de nouveau généré automatiquement pour chaque segment. Celui-ci est conservé dans le champ `nom` de l'objet `seg`. Le champ `bsident` contient le nom de la variable associée à la borne si celle-ci est fixe. Les quatre champs suivants conservent la position de la borne dans le graphe. Le champ `genreborne` indique si la borne est fixe ou non. Enfin, une liste de conditions est accessible à partir de `list_cond`. Chaque élément de cette liste est du type `expr_bool` déjà rencontré.

```
seg = class(Tobject)
      nom : string;
      bsident : string;
      hautx,hauty,basx,basy : integer;
      genreborne : Tgenreborne;
      list_cond : Plist_cond;
```

```
constructor create(_nom : string; _typeborne : Ttypeborne;
_bsconst : integer;_bsident : string ; _hautx,_hauty,_basx,
_basy : integer;_genreborne : Tgenreborne; _list_cond :
Plist_cond);
end;
```

6.1.2 Principes généraux

Dans cette section, nous donnons les principes généraux utilisés pour la construction de l'arbre syntaxique.

Nous avons défini une bibliothèque de procédures qui, pour chaque action réalisée sur la planche à dessin, a pour effet de modifier l'arbre syntaxique. C'est ainsi que l'arbre est généré. Au fur et à mesure que le dessin est produit par l'utilisateur, le système appelle les routines correspondant aux actions effectuées.

Par exemple, l'appel de la fonctionnalité "Ajout d'un tableau" a comme conséquence, si le processus s'est déroulé correctement, l'appel de la procédure "Ajoutertabl_ds_descript". L'appel de cette procédure ne sera pas réalisé si les valeurs entrées dans la boîte de dialogue ne sont pas valides ou si le dessin a été mal effectué.

Le tableau de la figure 6.1 reprend, pour chaque fonctionnalité de modification du dessin, la procédure qui est appelée si l'opération de modification a effectivement eu lieu.

Pour les conditions, nous procédons d'une autre manière puisqu'elles sont entrées sous forme d'une chaîne de caractères. Afin de générer la branche reprenant la condition entrée, nous avons utilisé le parseur dont nous disons quelques mots à la section suivante.

6.2 Arbre syntaxique des expressions et instructions

Cet arbre est construit par un parseur également implémenté sous Delphi. Pour le réaliser, nous nous sommes basés sur les notes [5] de Monsieur Le

Fonctionnalité	Procédure
Ajout d'un tableau	Ajoutertabl_ds_descript
Ajout d'une représentation	Ajouterrepres_ds_descript
Ajout d'une borne	Ajouterborne_ds_descript
Ajout d'une variable	Ajoutervar_ds_descript
Ajout d'une condition	Ajoutercond_ds_descript
Suppression d'un tableau	Supprimertabl_ds_descript
Suppression d'une représentation	Supprimerrepres_ds_descript
Suppression d'une borne	Supprimerborne_ds_descript
Suppression d'une variable	Supprimervar_ds_descript
Suppression d'une condition	Supprimercond_ds_descript

FIG. 6.1: les fonctionnalités de modification

Charlier. Chaque token qu'il reconnaît est placé dans un record de type `constr` dont voici la structure :

```
type chaine = array [1..8]of char;
     tbsymb = (identificateur,booleen,entier,specsymb,nosymb,errsymb);
     valtype = booleen..entier;
     message = string;
     tconstr = (simvar,indvar,unop,binop,cons,affect,cond1,cond2,
                tantque,debut,parent);
     operateur =(fois,quotient,plus,moins,egal,different,inferieur,
                 superieur,infeq,supeq,et,non,ou,inchange,permute,
                 tricrois,tristrcr,tristrde,tridecro,somme,produit,
                 min,max,pourtout,ilexiste,appart,egaltab,longueur);
     Pconstr = ^constr;
     Plconstr=^lconstr;

     constr = record
              case iconstr : tconstr of
                   debut : (linstr : Plconstr);
                   simvar : (vnom : chaine);
                   indvar : (tnom : chaine;
                             indice : Pconstr);
                   unop : (uop : operateur;
```

```
                oper : Pconstr);
        binop : (bop : operateur;
                 oper1, oper2 : Pconstr);
        cons : (constype : valtype;
                consval : integer);
        affect : (dexpr,expr : pconstr);
        cond1,cond2:(bexpr1,instr1,instr2:pconstr);
        tantque:(bexpr2,instr:pconstr);
        parent:(pexpr:pconstr)
end;
```

Suivant la valeur du champ `iconstr`, on pourra ensuite évaluer la sémantique des expressions et instructions. Le tableau ci-dessous reprend pour chaque valeur du champ `iconstr` l'objet syntaxique associé :

Valeur	Objet syntaxique
debut	bloc begin - end
simvar	variable simple
indvar	variable indicée
unop	opérateur unaire et argument
binop	opérateur binaire et arguments
cons	constante
affect	instruction d'affectation
cond1	instruction if then
cond2	instruction if then else
tantque	instruction while
parent	expression parenthésée

Nous listons maintenant les autres champs de l'objet `constr` et leur signification.

- Le champ `linstr` est un pointeur vers la liste des instructions du bloc begin-end. Cette liste est du type `lconstr` défini ci-dessous :

```
Plconstr= ^lconstr;
lconstr= record constrcour : Pconstr;
                listeconstr : Plconstr
         end;
```

- `vnom` contient un nom de variable.

- `tnom` contient un nom de tableau, le champ `indice` est un pointeur vers l'expression référençant un élément du tableau.
- Le champ `uop` indique le type d'opérateur unaire (-, not), `oper` est un pointeur vers l'argument de l'opérateur.
- `bop` donne le type d'opérateur binaire (+, -, and), `oper1, oper2` sont des pointeurs vers les arguments de l'opérateur.
- Le champ `constype` indique le type de la constante (entier, booléen), `consval` donne sa valeur.
- Les champs `dexpr` et `expr` sont respectivement des pointeurs vers la "left-value" et la "right-value".
- Le champ `bexpr1` pointe vers l'expression de l'instruction "if", `instr1` pointe vers l'instruction suivant le "then". `instr2` sert dans le cas d'une instruction "if then else" et pointe vers l'instruction suivant le "else".
- Le champ `bexpr2` est un pointeur vers la condition du "while", `instr` pointe vers l'instruction à effectuer lorsque la condition est vérifiée.
- Le champ `pexpr` pointe vers l'expression située à l'intérieur des parenthèses.

6.3 Sémantique des situations et instructions

La sémantique des situations est à distinguer de celle des instructions. En effet, la sémantique d'une situation ne peut prendre que deux valeurs différentes : vrai ou faux, la situation est vérifiée ou ne l'est pas. Par contre, la sémantique des instructions elle, a pour objectif de modifier les valeurs associées aux variables et tableaux donc, le store du programme.

La sémantique des situations a été implémentée à partir du développement théorique du chapitre 3. Lors de l'évaluation, nous supposons qu'une situation est vraie jusqu'à preuve du contraire. Nous évaluons d'abord les conditions de la situation, ensuite pour chaque tableau de la situation, nous évaluons leur sémantique.

La fonction évaluant la sémantique d'un tableau vérifie que chaque représentation est vraie. Dans chaque représentation, on vérifie que tous les

segments sont vrais. Et ainsi de suite jusqu'à avoir exploré toutes les ramifications.

Le tableau ci-dessous reprend les procédures permettant de calculer la sémantique des situations et pour chacune de celle-ci, son but et la procédure appelante.

Nom	Procédure appelante	But
sem_sit		évalue la sémantique d'une situation
sem_expr	sem_sit	évalue la sémantique d'une condition
sem_tabl	sem_sit	évalue la sémantique d'un tableau
sem_repres	sem_tabl	évalue la sémantique d'une représentation
sem_bloc_i	sem_repres	évalue la sémantique d'un bloc insécable
sem_bloc_s1	sem_repres	évalue la sémantique d'un bloc sécable de type 1
sem_bloc_sp	sem_repres	initialise certains paramètres et appelle sem_bloc_sp2
sem_bloc_sp2	sem_bloc_sp	évalue récursivement la sémantique d'un bloc sécable de type 2
sem_segment	sem_bloc_i, sem_bloc_s1, sem_bloc_sp2	évalue la sémantique d'un segment
sem_expr	sem_segment	évalue la sémantique d'une condition

Quant à la sémantique des instructions, disons simplement qu'elle vérifie celle du Pascal standard. Nous avons également écrit des routines vérifiant que tous les objets utilisés dans le code de l'utilisateur ont bien été déclarés.

La déclaration des variables et tableaux s'effectue grâce au menu de l'interface et non dans le code du programme. Il aurait été intéressant de permettre à l'étudiant de déclarer les objets qu'il manipule au sein de son propre programme.

6.4 Evaluation de Delphi

Delphi est un environnement de travail pour le développement d'applications Windows. Il comprend un langage de programmation orienté objet qui est une évolution du Borland Pascal 7.0. Il permet la création plus aisée d'applications interactives. Pour ce faire, il comporte une large bibliothèque d'objets graphiques prédéfinis. En outre, elle possède des méthodes manipulant d'autres objets telles que des méthodes de gestion de listes.

L'avantage principal de Delphi est qu'il évite au programmeur de devoir implémenter les fenêtres et autres objets d'interaction (boutons, listes de sélection, menus, ...).

D'autre part, il est surprenant qu'un logiciel comprenant un ensemble de fonctionnalités aussi vaste possède une aide en ligne aussi peu détaillée. Aussi, le module de dessin ne nous a pas semblé très stable ni très clair. La pauvreté de la documentation en est peut-être la cause. Enfin, Delphi est utile dans le cas d'un développement d'un programme interactif mais se révèle trop rigide pour un programme sans interaction ; par exemple, tout traitement est déclenché par un événement (clic d'un bouton).

Chapitre 7

Les critiques

Le but de ce chapitre est de dégager les faiblesses du système et de dégager les moyens d'y remédier.

7.1 Fonctions de sauvegarde

Le système devrait être capable de sauvegarder les situations décrites et le code entré. De cette manière, l'étudiant pourrait interrompre son travail ; l'enseignant pourrait lui fournir des solutions aux énoncés.

7.2 Interface pour la saisie des expressions

Bien que cela fasse partie du cahier des charges, nous n'avons pas implémenté une interface W.Y.S.I.W.Y.M (What You See Is What You Mean) pour la saisie des expressions. Néanmoins, la représentation graphique des conditions dans le dessin risquerait d'être illisible si le segment les contenant était trop réduit. La solution (un symbole marquant la présence d'une ou de plusieurs conditions) adoptée pour le moment semble être un compromis idéal entre représentativité et légèreté.

7.3 Validation du code

Le système, actuellement, permet de vérifier que tous les objets ont été créés à l'aide des menus "Ajouter variable" et "Ajouter tableau". Maintenant,

il faudrait ajouter une syntaxe de déclaration au parseur, l'utilisateur pourrait déclarer les variables et tableau comme il le ferait dans un programme Pascal. De plus, l'analyseur syntaxique devrait effectuer un "type-checking" sur l'ensemble du code. Ceci éviterait, par exemple, que l'étudiant construise une instruction conditionnelle à partir d'une expression entière.

7.4 Interface graphique

Un inconvénient majeur de l'interface graphique actuelle est qu'une borne fixe désigne uniquement, l'élément la précédant. Une évolution intéressante serait de permettre à cette borne de désigner une des deux cellules adjacentes à celle-ci, cela permettrait une utilisation plus souple du système. Il faudrait modifier la syntaxe du langage graphique afin de pouvoir relier deux segments au sein d'une même condition. On éviterait ainsi le recours aux variables auxilliaires de l'exemple 3 du chapitre 5.

7.5 Initialisation automatique de contenu

Une fonctionnalité supplémentaire devrait être l'initialisation automatique du contenu des tableaux et des valeurs des variables. Ce qui éviterait, sans doute, que l'utilisateur introduise des valeurs qui vérifieraient systématiquement les relations 1.2.3. Cette fonctionnalité devrait être capable de générer des exemples "limites" se basant sur les conditions énoncées.

7.6 Bibliothèque de problèmes

Il serait intéressant que le professeur mette à la disposition des étudiants un ensemble de problèmes à résoudre. Cet ensemble pourrait se décomposer en deux catégories :

1. Le problème est simplement posé et l'étudiant doit produire le code et les spécifications ;
2. Le problème et les spécifications sont énoncés, l'étudiant doit produire le code correspondant.

Conclusion

Le but de ce travail était de développer un outil d'aide à l'apprentissage de la programmation. Celui-ci se basait sur la méthode proposée par Monsieur Le Charlier dans son article "Un système d'aide à l'enseignement d'une méthode de programmation" [1]. Dans celle-ci, un problème est décrit par trois situations. Elles énoncent des conditions sur les variables et tableaux du programme à réaliser. Le principe de la méthode est de vérifier la concordance entre le code et les situations décrites.

Pour ce faire, nous avons dû spécifier et implémenter un langage graphique. Celui-ci permet de décrire des conditions portant sur des variables et des tableaux. Ensuite, nous avons écrit un parseur d'un sous-langage Pascal. L'implémentation du langage graphique et ce parseur nous ont permis de stocker les situations et le code d'un programme. Grâce à l'écriture et l'implémentation de la sémantique du langage graphique, nous étions en mesure de confronter la description du programme et ses instructions. Nous avons rencontré des limitations dans l'utilisation de notre langage comme le montre le troisième exemple exposé.

Les améliorations que nous avons envisagées par après mais que nous n'avons pu réaliser faute de temps, se trouvent dans le chapitre 7. Nous les reprenons succinctement ci-dessous :

- l'implémentation de fonctions de sauvegarde,
- l'implémentation d'une interface pour la saisie des expressions,
- l'ajout d'une syntaxe de déclaration au parseur,
- la vérification des types,
- l'initialisation automatique des objets.

Bibliographie

[1] M. Derroitte and B. Le Charlier. Une Système d'aide à l'Enseignement d'une Méthode de Programmation. In J. Arsac, editor, *Actes du Premier Colloque Francophone sur la Didactique de l'Informatique*, Paris, Juillet 1989. EPI.

[2] M. De Wolf. Aide à l'apprentissage de la programmation : Interpréteur pour un langage algorithmique avec assertions. Technical report, UCL, 1994.

[3] N. Habra. Paradigmes de programmation. Cours de première licence informatique, 1996.

[4] L. Lamport. *LaTeX : A Document Preparation System.* Addison-Wesley, 1986.

[5] B. Le Charlier. Définition et implémentation du langage lsd80. 1980.

[6] J. Rulkin. Contribution à l'implémentation d'un langage graphique de description d'assertions. Master's thesis, FUNDP, 1989.

[7] L. Simon. Réalisation d'un programme de vérification des invariants. Master's thesis, FUNDP, 1992.

[8] J. Vanderdonckt. Conception d'interface homme-machine. Cours de deuxième licence informatique, 1997.